Foundations of Computing & Programming

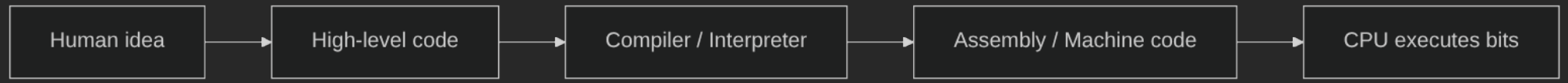
Python Course · Module 1

ElnurBDa

What is Programming?

Programming is the process of giving instructions to a computer in a language it understands.

Computers themselves only understand **machine code**, so programming languages allow humans to write instructions more easily.



Think of it as translation layers that convert human steps into electrical signals.

Programming is used for:

- Automation
- Web development
- **Cybersecurity**
- Data analysis
- System administration

Real-world examples:

- ATMs
- Browsers
- Mobile apps
- Network scanners (Nmap, etc.)

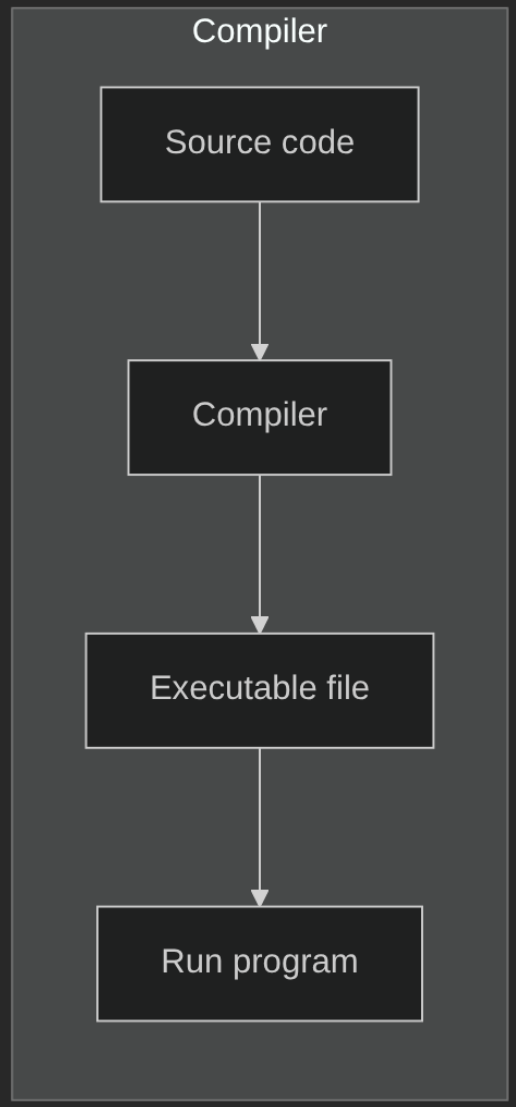
Why Programming Matters

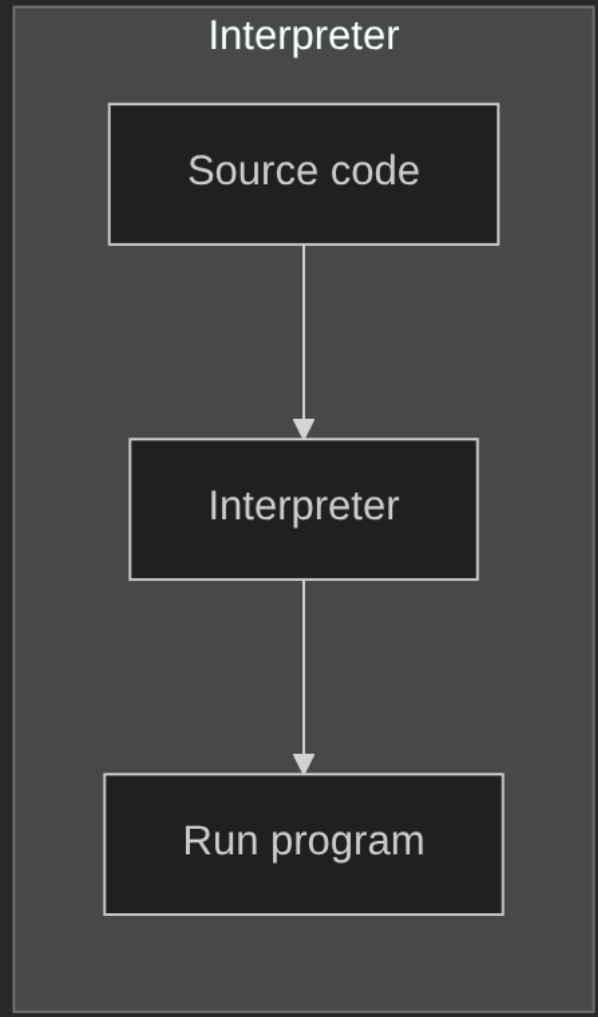
Programming allows you to:

- Solve problems
- Automate repetitive tasks
- Build software
- Understand how computers actually work
- Strengthen cybersecurity skills (scripting, automation)

It is a foundational skill for modern IT and cybersecurity professionals.

Compilers vs Interpreters





Compiler

A **compiler** converts the entire source code into machine code **before** the program runs.

Characteristics:

- Produces a standalone executable
- Faster execution
- Errors found before running

Examples:

- C
- C++
- Go
- Rust

▒▒ Example:

hello.c

```c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

```sh
# compile
gcc hello.c -o hello
# run
chmod +x hello
./hello
# Result: Hello, world!
```

Interpreter

An **interpreter** executes code **line by line**.

Characteristics:

- More flexible
- Easier to test small parts
- Slower execution
- Errors show up while running

Examples:

- Python
- JavaScript
- Ruby

▒▒ Example:

hello.py

```
print("Hello, world!")
```

```
# run
python hello.py
# Result: Hello, world!
```

Algorithms – Introduction

An **algorithm** is a step-by-step procedure to solve a problem.

Characteristics of good algorithms:

- Clarity
- Efficiency
- Consistency
- Finite steps

Common examples in daily life:

- Sorting items
- Making tea
- Following a recipe
- Unlocking your phone

Algorithm Examples

Real-life algorithm example (Making Tea):

1. Boil water
2. Place tea bag in cup
3. Pour water
4. Wait 3–5 minutes
5. Remove tea bag
6. Drink

Pseudocode example:

```
input number
if number > 0:
    print("Positive")
else:
    print("Non-positive")
```

Create a Simple Algorithm – Activity

Write an algorithm for:

1. Logging into a website
2. Buying a product online
3. Starting a computer

Goal: describe steps **clearly** so that even a computer could follow them.

Algorithm Design Checklist

Step	Questions to ask
Inputs	What data do I need? What format?
Outputs	What should be produced?
Steps	In what order? Any loops or decisions?
Edge cases	What if input is missing/invalid?
Representation	Pseudocode, flowchart, or plain text?

Comparison:

- **Pseudocode** — quick, language-agnostic
- **Flowchart** — great for visual branching
- **Plain text** — fast brainstorming, less structure

Installing Python

Check whether Python is installed:

```
python3 --version
```

Install on Linux (Debian/Ubuntu):

```
sudo apt update
sudo apt install python3
```

Install on macOS (Homebrew):

```
brew install python
```

Install on Windows:

- Use Microsoft Store
- Or download installer from python.org

Enter interactive mode (REPL):

```
python3
```

Exit REPL with:

```
exit()
# or press Ctrl+D on Linux/macOS
```

Running Python Code

Run a script file:

```
python3 script.py
```

Simple script example (hello.py):

```
print("Hello, world!")
```

Then run:

```
python3 hello.py
```

Environment & Workflow Lab

Python Course · Module 1.5

ElnurBDa

Module 1.5 — Environment & Workflow Lab

This short module is a **hands-on lab** between Module 1 and Module 2.

Goals:

- Set up a **consistent workspace** for the rest of the course
- Practice running Python files from the **terminal**
- Create and activate a **virtual environment (venv)**
- Understand where your scripts and data files live on disk

After this lab, Module 2 (Python basics) will feel much more natural.

Course Folder Layout

Pick a main folder where all course projects will live, for example:

```
/home/elnur/Desktop/idschool/python-course/
├── markdown_files/   # slides
└── projects/         # your code lives here
```

Inside `projects/` you will create one folder per module or mini-project:

```
projects/
├── module1_hello/
├── module2_basics/
├── module3_data/
└── ...
```

Try to keep code, data files, and virtual environments **inside** these project folders.

Terminal Basics Refresher

Open a terminal (Linux/macOS Terminal, PowerShell, etc.) and practice:

```
pwd            # print current directory
ls             # list files (dir on Windows)
cd path/to/dir # change directory
```

Navigation checklist:

- Can you navigate from your home folder to `idschool/python-course/`?
- Can you list the contents of `markdown_files/` and `projects/`?

You should **always know where you are** before running `python3 file.py`.

Creating Your First Project Folder

Let’s create `module1_hello`:

```
cd /home/elnur/Desktop/idschool/python-course
mkdir -p projects/module1_hello
cd projects/module1_hello
```

Create `hello.py` in this folder with the following content:

```
print("Hello from Module 1.5!")
```

Run it:

```
python3 hello.py
```

If you see the message, your basic workflow is working.

Virtual Environments — Why Now?

Later modules will install packages like requests, Flask, etc.

Without virtual environments:

- Different projects may **fight over versions**
- You may not remember which scripts require which packages

With virtual environments:

- Each project can have its **own dependencies**
- You can safely experiment without breaking other projects

We start using this pattern now so it feels natural later.

Creating and Activating a venv

Inside `projects/module1_hello/`:

```
python3 -m venv .venv
```

Activate it:

```
source .venv/bin/activate   # Linux/macOS
# or on Windows (PowerShell):
# .venv\Scripts\Activate.ps1
```

Your prompt should now show something like:

```
(.venv) user@machine:~/.../module1_hello$
```

To deactivate later:

```
deactivate
```

Installing a Test Package

While still inside the (.venv):

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

You should see requests and its dependencies listed.

Optional: save them to requirements.txt:

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

Now this project knows exactly **which versions** it uses.

Using pathlib for Safer Paths

To avoid hard-coded string paths, use pathlib:

```
from pathlib import Path

BASE_DIR = Path(__file__).parent  # folder containing this file
data_dir = BASE_DIR / "data"

print("Base dir:", BASE_DIR)
print("Data dir:", data_dir)
```

———————————————— [finished] ————————————————

```
Base dir: /tmp/nix-shell-61594-0/.presentermXx6S8w
Data dir: /tmp/nix-shell-61594-0/.presentermXx6S8w/data
```

Pattern:

- `Path(__file__).parent` gives you the current project folder
- `BASE_DIR / "subfolder" / "file.txt"` builds paths safely

We will reuse this idea when working with files, logs, and web servers.

Python Basics

Python Course · Module 2

ElnurBDa

Variables

A variable stores a **reference** to a value.

Created by assignment:

```
name = "Alice"
age = 20
height = 1.75
```

Rules:

- Start with a letter or _
- Letters, digits, underscores allowed
- Case-sensitive: count ≠ Count

Data Types

▒▒▒ int

Whole numbers, positive or negative.
Example:

```
age = 25
```

▒▒▒ float

Decimal numbers.

```
pi = 3.14
```

▒▒▒ str

Text data in quotes.

```
city = "Baku"
```

▒▒▒ bool

Logical values: True or False.

```
is_student = True
```

▒▒▒ NoneType

Represents “no value”.

```
result = None
```

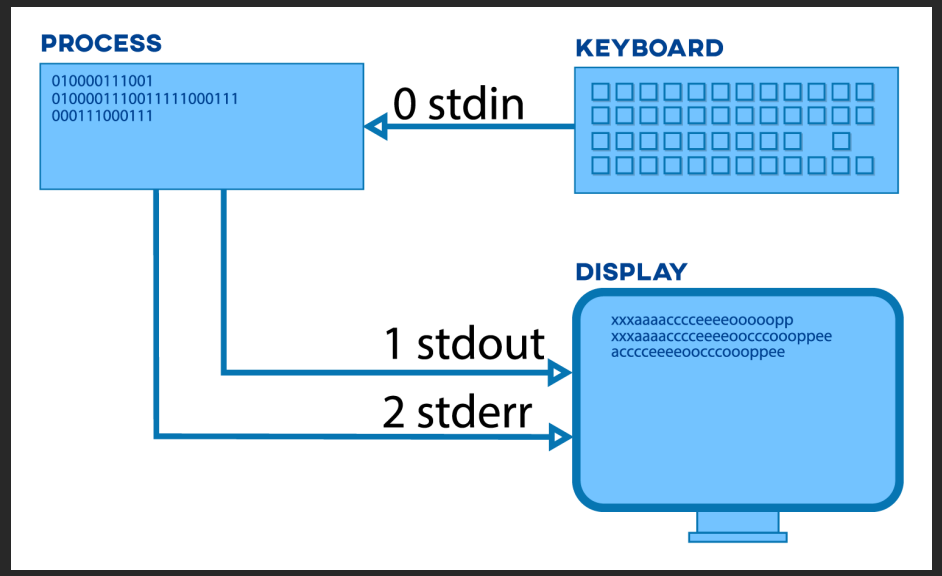
Input & Output

▒▒ Output (stdout)

```
name = "ElnurBDa"
print("Hello")
print("User:", name)
print(f"The F String is used by
{name}")
```

```
──────────── [finished] ────────────
```

```
Hello
User: ElnurBDa
The F String is used by ElnurBDa
```



▒▒ Input (stdin)

```
name = input("Enter your name: ")
```

`input()` **always** returns `str`.

Convert for numeric use:

```
age = int(input("Enter age: "))
```

Arithmetic Operators

Operator Summary

```
+   addition
-   subtraction
*   multiplication
/   division (float)
//  floor division (drops decimals)
%   remainder
**  exponent
```

Examples

```
10 + 3   # 13
10 - 3   # 7
10 * 3   # 30
10 / 3   # 3.333...
10 // 3  # 3
10 % 3   # 1
2 ** 3   # 8
```

Comparison Operators

```
x == y   # equal
x != y   # not equal
x > y
x < y
x >= y
x <= y
```

Example:

```
score = 85
score >= 90   # False
score < 100   # True
```

Logical Operators

```
and   # both conditions True
or    # at least one True
not   # reverses boolean
```

Example:

```
age = 20
has_id = True

age >= 18 and has_id   # True
age < 18 or has_id     # True
not has_id             # False
```

Assignment Operators

Shorthand updates:

```
x = 5
x += 2   # 7
x -= 1   # 6
x *= 3   # 18
x /= 2   # 9.0
```

Used often in loops and accumulations.

Type Casting (Conversion)

Convert between types:

```
int("10")      # 10
float("3.14")  # 3.14
str(123)       # "123"
bool("hi")     # True
bool("")       # False
```

Used for:

- numeric input
- formatting
- arithmetic

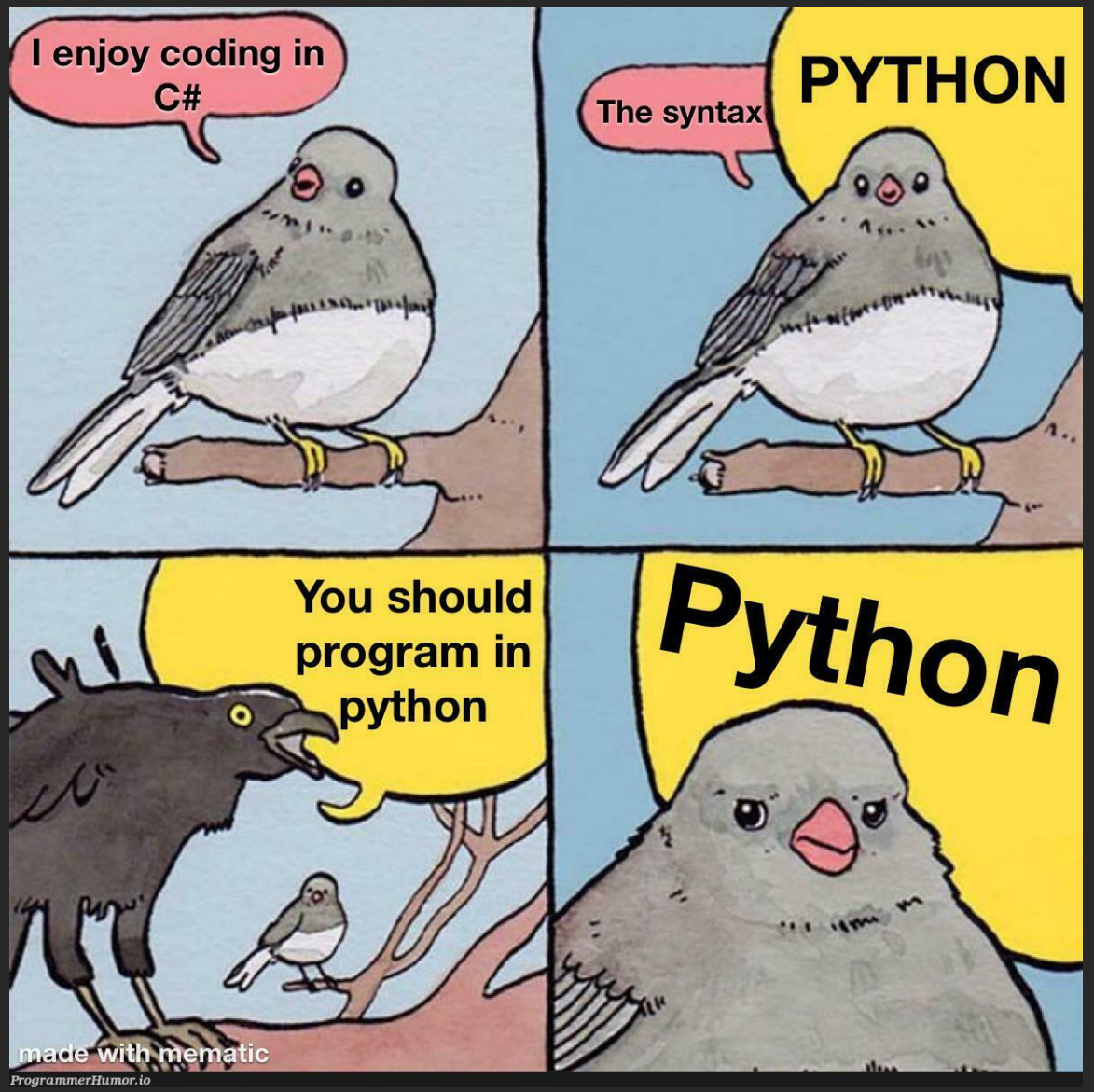
Example

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))

next_age = age + 1
is_adult = age >= 18
message = f"{name}, next year you will be {next_age}."

print(message)
print("Adult status:", is_adult)
```

Mini Task

Create a script that:

1. Asks for two numbers
2. Converts them to integers
3. Calculates:
   - sum
   - difference
   - product
4. Prints results



Python Basics Practice

Python Course · Module 2 Practice

ElnurBDa

Module 2 Practice — Variables, Types & Operators

This practice module sits between Modules 2 and 3.

Goals:

- Get more **hands-on** with variables and types
- Become fluent with **input/output** and **type casting**
- Practice arithmetic, comparison, and logical operators
- Prepare your brain for data structures in Module 3

Use this as a **lab deck**: lots of short exercises, less theory.

Warm-Up — Quick Questions

Without running code, guess the results:

```
1 + 2 * 3
(1 + 2) * 3
10 / 3
10 // 3
10 % 3
```

Then open a Python REPL or script file and **verify** your answers.

Why? Getting a “feel” for operator behavior makes later code easier to read.

Practice 1 — Simple Calculator

Task:

Create `calculator.py` inside `projects/module2_basics/` that:

1. Asks the user for two numbers.
2. Converts them to `float`.
3. Computes and prints:
   - sum
   - difference
   - product
   - quotient (second not zero)

Example interaction:

```
Enter first number: 10
Enter second number: 3
Sum: 13.0
Difference: 7.0
Product: 30.0
Quotient: 3.3333333333
```

Try adding **rounding** with `round(value, 2)`.

Practice 2 — BMI Calculator

Create `bmi.py` that:

1. Asks for weight in kg (`float`)
2. Asks for height in meters (`float`)
3. Computes BMI = weight / (height ** 2)
4. Prints BMI rounded to 1 decimal place

Add basic classification (just using if/elif):

- BMI < 18.5 → “Underweight”
- 18.5 ≤ BMI < 25 → “Normal”
- 25 ≤ BMI < 30 → “Overweight”
- ≥ 30 → “Obese”

This connects arithmetic, comparison operators, and string formatting.

Practice 3 — String Playground

In strings_play.py:

1. Ask for the user's full name (one line of text).
2. Print:
   - Length of the string
   - Uppercase version
   - Lowercase version
   - First and last character (if not empty)
3. Create a **short username** suggestion:
   - first 3 letters of the name (no spaces)
   - plus the string length

Example:

```
Full name: Ada Lovelace
Length: 12
Upper: ADA LOVELACE
Lower: ada lovelace
Short username: Ada12
```

Practice 4 — Truth Tables with and / or / not

In logic_table.py, without using user input:

1. Create variables a = True, b = False.
2. Print a simple truth table:

```
a      b      a and b   a or b   not a
True   False  False     True     False
...
```

Use **f-strings** for clean formatting.

Goal: see how logical operators behave for all combinations.

Practice 5 — Age Gate

`age_gate.py`:

1. Ask user for age (convert to `int`).
2. Use comparisons and `and`/`or` to decide:
   - `< 13` → “Child account”
   - `13–17` → “Teen account”
   - `18+` → “Adult account”
3. Use a boolean variable `has_parent_consent` (True/False) and update messages:
   - If `< 18` and no consent → “Access denied”

This will warm you up for more structured `if` logic and loops later.

Practice 6 — Simple Tip Calculator

`tip.py`:

1. Ask for bill amount (`float`).
2. Ask for desired tip percentage (e.g., 10, 15, 20).
3. Compute tip and total.
4. Print results with 2 decimal places.

Extra: allow the user to input tip as either `0.15` or `15`, and handle both correctly.

Mini Project — Profile Summary

Create `profile.py` that:

1. Asks for:
   - name (`str`)
   - age (`int`)
   - city (`str`)
   - favorite number (`int`)
2. Uses arithmetic and logic to compute:
   - age next year
   - whether the favorite number is even or odd
3. Prints a nicely formatted multi-line summary using an f-string.

Example output:

```
Hello, Ada!
You live in London and next year you will be 37.
Your favorite number 42 is even.
```

This project consolidates most of Module 2 concepts before you meet lists and dicts.

Data Structures

Python Course · Module 3

ElnurBDa

Data Structures – Overview

Python provides multiple built-in structures for organizing data:

- **Lists**
- **Strings**
- **Tuples**
- **Sets**
- **Dictionaries**
- Choosing the right structure

Each structure has unique properties and use-cases.

Lists

Lists store **ordered, changeable** sequences.

```
numbers = [10, 20, 30]

# Operations:
numbers.append(40)
numbers[1] = 99
numbers.remove(10)
print(numbers)

# List indexing:
print(numbers[0])
print(numbers[-1])
print(numbers[1:3])
```

———————— [finished] ————————

```
[99, 30, 40]
99
40
[30, 40]
```

List Comprehensions

List comprehensions are a **compact way** to build new lists from existing data.

Basic pattern:

```
numbers = [1, 2, 3, 4, 5]

squares = [n * n for n in numbers]
evens = [n for n in numbers if n % 2 == 0]

print("squares:", squares)
print("evens:", evens)
```

———————————— [finished] ————————————

```
squares: [1, 4, 9, 16, 25]
evens: [2, 4]
```

They are great for:

- simple transformations
- filtering values
- avoiding manual `for` + `.append()` boilerplate

Rule of thumb: if the logic fits in **one short line**, a comprehension is OK; otherwise, use a normal loop for readability.

List Characteristics

Advantages:

- Mutable
- Dynamic size
- Supports duplicates

Use-cases:

- Collections
- Queues
- Storing ordered items

Strings

Strings represent **text**.

Properties:

- Ordered
- Immutable
- Indexable
- Iterable

```
s = "Hello"

a = s[0]       # first character
b = s[-1]      # last character
c = s.upper()  # all upper chars
d = len(s)     # Length of the string

print(a, b, c, d)
```

```
──────────────────── [finished] ────────────────────
```

```
H o HELLO 5
```

Tuples

Tuples store **ordered, immutable** sequences.

```
point = (10, 20)
```

Benefits:

- Fast
- Secure from modification
- Memory-efficient

Tuple unpacking:

```
x, y = point
```

Tuple vs List

- Need modification → **List**
- Need fixed, reliable structure → **Tuple**
- Performance required → **Tuple**

Sets

Sets store **unordered, unique** elements.

```
colors = {"red", "blue", "green"}
```

Operations:

```
colors.add("yellow")
colors.remove("red")
```

Useful for:

- membership checks
- removing duplicates
- mathematical operations

Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}

x1 = a | b    # union
x2 = a & b    # intersection
x3 = a - b    # difference
x4 = a ^ b    # symmetric difference

print(x1, x2, x3, x4)
```

———————— [finished] ————————

```
{1, 2, 3, 4, 5} {3} {1, 2} {1, 2, 4, 5}
```

Dictionaries

Dictionaries store **key : value** pairs.

```
user = {
    "name": "Alice",
    "age": 25,
    "city": "Baku"
}

# Access:
n = user["name"]
print(n)
print()

# Modify:
user["age"] = 26
user["country"] = "Azerbaijan"
print(user)
```

———————————— [finished] ————————————

```
Alice

{'name': 'Alice', 'age': 26, 'city': 'Baku', 'country':
'Azerbaijan'}
```

Dictionary Operations

```
user = {
    "name": "Alice",
    "age": 25,
    "city": "Baku"
}

a = user.keys()
b = user.values()
c = user.items()
d = "city" in user  # check key existence
e = "Baku" in user

print(a)
print(b)
print(c)
print(d)
print(e)
```

```
———————————————— [finished] ————————————————
```

```
dict_keys(['name', 'age', 'city'])
dict_values(['Alice', 25, 'Baku'])
dict_items([('name', 'Alice'), ('age', 25), ('city',
'Baku')])
True
False
```

Dictionaries are ideal for structured data.

Example

```
student = {
    "name": "Elnur",
    "scores": [90, 85, 97],
    "passed": True
}

average = sum(student["scores"]) / len(student["scores"])

print(student["name"], "average:", average)
```

———————————————— [finished] ————————————————

```
Elnur average: 90.66666666666667
```

Choosing the Right Data Structure

Type	Ordered	Mutable	Unique	Best For
String	Yes	No	N/A	Text
List	Yes	Yes	No	General collections
Tuple	Yes	No	No	Fixed data
Set	No	Yes	Yes	Unique items
Dict	Keys no	Yes	Keys unique	Structured data

Mini Task

Create a script that:

1. Stores student info in a dictionary
2. Contains: name, age, list of grades
3. Calculates average grade
4. Prints a formatted summary

Control Flow

Python Course · Module 4

ElnurBDa

Control Flow — Overview

Control flow decides **how a program behaves**:

- Conditional statements
- Logical operators
- Loops (`for`, `while`)
- Loop control (`break`, `continue`, `pass`)

This module teaches how Python makes decisions and repeats actions.

Conditional Statements

Basic structure:

```
if condition:
    block
```

A condition must evaluate to **True** or **False**.

Example:

```
age = 20
if age >= 18:
    print("Adult")
```

———————————— [finished] ————————————

```
Adult
```

---

if / else

```
if condition:
    action_if_true
else:
    action_if_false
```

Example:

```
temperature = 10

if temperature > 15:
    print("Warm")
else:
    print("Cold")
```

———————— [finished] ————————

```
Cold
```

if / elif / else

Used when multiple conditions must be checked in order.

```
grade = "?"
score = 75

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
else:
    grade = "D"

print(grade)
```

———————— [finished] ————————

```
C
```

Logical Operators in Conditions

```
and   # both True
or    # one True
not   # invert
```

Examples:

```
age = 20
has_id = True

age >= 18 and has_id
age < 18 or has_id
not has_id
```

Nested Conditions

```
age = 25
citizen = True

if age >= 18:
    if citizen:
        print("Eligible")
    else:
        print("Not eligible")
```

———————————— [finished] ————————————

```
Eligible
```

Avoid deep nesting when possible.

---

for Loop — Basics

Iterates over a sequence:

```
for x in [1, 2, 3]:
    print(x)
```

———————————— [finished] ————————————

```
1
2
3
```

Strings:

```
for char in "Python":
    print(char)
```

———————————— [finished] ————————————

```
P
y
t
h
o
n
```

---

for Loop with range()

```
for i in range(5):
    print(i)
```

```
———————————————— [finished] ————————————————
```

```
0
1
2
3
4
```

Custom range:

```
for i in range(2, 10, 2):
    print(i)
```

```
———————————————— [finished] ————————————————
```

```
2
4
6
8
```

---

while Loop

Repeats while a condition is **True**.

```
count = 0
while count < 5:
    print(count)
    count += 1
```

———————————— [finished] ————————————

```
0
1
2
3
4
```

Loop Control — break

Stops the loop immediately.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

———————————— [finished] ————————————

```
0
1
2
3
4
```

Loop Control — continue

Skips current iteration.

```
for i in range(5):
    if i % 2 == 0:
        continue
    print(i)
```

———————————— [finished] ————————————

```
1
3
```

Loop Control — pass

`pass` does nothing — placeholder.

```
for i in range(3):
    pass
```

──────────────── [finished] ────────────────

Loop Patterns & Comprehensions

Often we use loops to **build new lists** from old ones.

Classic pattern:

```
nums = [1, 2, 3, 4, 5]
evens = []

for n in nums:
    if n % 2 == 0:
        evens.append(n)

print("evens:", evens)
```

———————— [finished] ————————

```
evens: [2, 4]
```

Same logic using a **list comprehension** (from Module 3):

```
nums = [1, 2, 3, 4, 5]
evens = [n for n in nums if n % 2 == 0]
print("evens:", evens)
```

———————— [finished] ————————

```
evens: [2, 4]
```

Use whichever is **clearer** to you and your team; for multiple complex steps, prefer the classic loop.

Example

```python
total = 0

for i in range(1, 6):
    if i % 2 == 0:
        total += i
    else:
        continue

print("Sum of even numbers:", total)
```

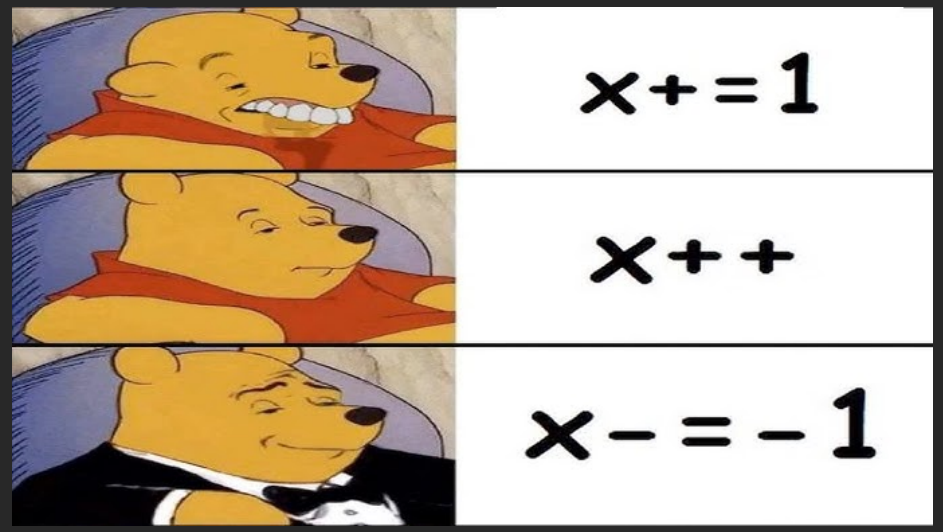
———————————— [finished] ————————————

```
Sum of even numbers: 6
```

Mini Task

Write a script that:

1. Asks the user for a number
2. Prints all numbers from 1 to that number
3. Prints only **even numbers**
4. Skips odd numbers using `continue`
5. Stops entirely if number exceeds 100 using `break`



Refactoring to Functions

Python Course · Module 4.5

ElnurBDa

Module 4.5 — From Loops to Reusable Functions

Goals:

- Recognize when repeated patterns should become **functions**
- Practice extracting logic from loops into named helpers
- Prepare for multi-file organization in Module 5

Think of this as “cleaning up” your scripts before modularizing them.

Why Refactor to Functions?

Example of a script **without** functions:

```
scores = [80, 95, 60, 70]

total = 0
for s in scores:
    total += s
average = total / len(scores)
print("Average:", average)

passed = 0
for s in scores:
    if s >= 70:
        passed += 1
print("Passed:", passed)
```

Problems:

- Harder to reuse logic with other lists
- No clear names for key operations

We'll refactor step by step.

Step 1 — Wrap Logic in a Function

```python
def average(scores):
    total = 0
    for s in scores:
        total += s
    return total / len(scores)

scores = [80, 95, 60, 70]
print("Average:", average(scores))
```

———————————— [finished] ————————————

```
Average: 76.25
```

Benefits:

- Name `average` describes intent
- Easy to test with different lists

Try writing `count_passed(scores, threshold)` in the same style.

Step 2 — Use Existing Constructs

After writing the basic function with a loop, you can improve it:

```
def average(scores):
    return sum(scores) / len(scores)

def count_passed(scores, threshold=70):
    passed = 0
    for s in scores:
        if s >= threshold:
            passed += 1
    return passed

scores = [80, 95, 60, 70]
print("Average:", average(scores))
print("Passed:", count_passed(scores))
```

———————— [finished] ————————

```
Average: 76.25
Passed: 3
```

Pattern:

1. Make it work
2. Make it **clear**
3. Only then make it shorter if it’s still readable

Mini Practice — Refactor Temperature Script

Take a plain script like:

```
temps = [20, 25, 18, 30]

for t in temps:
    if t > 25:
        print(t, "Hot")
    else:
        print(t, "OK")
```

Refactor into:

- `label_temp(t)` → returns `"Hot"` or `"OK"`
- `print_labeled_temps(temps)` → loops and prints

Then call these functions from `if __name__ == "__main__":`.

Separating Concerns — Input vs Logic vs Output

Good structure:

```python
def read_scores():
    raw = input("Enter scores separated by commas: ")
    parts = raw.split(",")
    return [int(p.strip()) for p in parts]

def compute_average(scores):
    return sum(scores) / len(scores)

def main():
    scores = read_scores()
    avg = compute_average(scores)
    print("Average:", avg)

if __name__ == "__main__":
    main()
```

———————————— [finished with error] ————————————

```
Enter scores separated by commas: Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermF5X1aP/snippet.py", line 15, in
<module>
    main()
    ~~~~^^
  File "/tmp/nix-shell-61594-0/.presentermF5X1aP/snippet.py", line 10, in main
    scores = read_scores()
  File "/tmp/nix-shell-61594-0/.presentermF5X1aP/snippet.py", line 2, in
read_scores
    raw = input("Enter scores separated by commas: ")
EOFError: EOF when reading a line
```

Roles:

- `read_scores` → input parsing
- `compute_average` → pure calculation
- `main` → wiring everything together

This pattern is exactly what Module 5 will build on.

Mini Project — Number Analyzer

Create `number_analyzer.py` that:

1. Has a `read_numbers()` function that reads a comma-separated line and returns a list of `int`.
2. Has `summary_stats(numbers)` that returns a dict with:
   - `count`, `min`, `max`, `average`
3. Has `print_report(stats)` that prints a formatted summary.
4. Uses a `main()` function plus the `if __name__ == "__main__":` pattern.

You'll reuse this style when you move to multiple files and modules next.

Functions & Modular Code

Python Course · Module 5

ElnurBDa

Functions — Introduction

Functions group reusable code into named blocks.

Benefits:

- Avoid repetition
- Improve readability
- Easier debugging
- Modular structure

Basic structure:

```
def name():
    pass
```

Creating Functions

```
def greet():
    print("Hello!")
```

Call:

```
greet()
```

Functions must be defined **before** calling.

Parameters

```
def greet(name):
    print("Hello,", name)
```

Multiple parameters:

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

Return Values

```
def square(x):
    return x * x
```

Functions without `return` return `None`.

Multiple Returns

```
def values():
    return 10, 20, 30
```

Unpacking:

```
a, b, c = values()
```

Variable Scope

```
x = 10      # global

def func():
    y = 5   # local
```

Local variables override global ones inside functions.

---

global Keyword

```
count = 0

def inc():
    global count
    count += 1
```

Use sparingly; global state makes code harder to manage.

Lambda Functions

```
square = lambda x: x * x
```

Often used inline:

```
add = lambda a, b: a + b
```

Higher-Order Functions

```
nums = [1, 2, 3, 4]

squares = list(map(lambda x: x*x, nums))
```

Using normal function:

```
def sq(x): return x*x
```

Modules — Importing

```
import math
math.sqrt(16)
```

Selective:

```
from math import sqrt
```

Rename:

```
import math as m
```

Built-in Modules

Examples:

- math
- random
- os
- sys
- datetime

Example:

```
import random
x = random.randint(1, 10)
```

Creating Your Own Module

File:

```
mymath.py
```

Content:

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

Usage:

```
import mymath
mymath.add(2, 3)
```

Organizing Code

```
project/
├── main.py
├── utils.py
└── math_ops.py
```

```
from utils import greet
from math_ops import add
```

Example

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

# math_ops.py
def multiply(a, b):
    return a * b

# main.py
from utils import greet
from math_ops import multiply

print(greet("Elnur"))
print(multiply(4, 5))
```

Mini Task

Create a project with:

1. main.py
2. converter.py with:
   - km_to_miles(km)
   - celsius_to_fahrenheit(c)
3. Import into main.py
4. Ask user input and print converted values

Objects & Classes — Why They Matter

Functions are perfect for **single actions**. Classes help when you need to model **entities** (users, sensors, tickets) that carry both data and behavior.

- **Object** = “thing” with attributes + methods
- **Class** = blueprint describing what every object of that type should have

Analogy:

- Class = recipe for User
- Object = actual user baked from that recipe

Use classes when you keep passing the same fields around or when multiple behaviors belong to the same piece of data.

Defining a Class

```python
class User:
    def __init__(self, username, email):
        self.username = username
        self.email = email
        self.active = True

    def deactivate(self):
        self.active = False

user = User("alice", "alice@example.com")
print(user.username, user.active)
user.deactivate()
print(user.active)
```

———————— [finished] ————————

```
alice True
False
```

Notes:

- `__init__` runs every time you call `User(...)`
- `self` points to the current object so you can store/read attributes
- Methods are normal functions defined inside the class

Attributes vs Methods

Concept	Purpose	Example
Attribute	Data stored on the object	`user.email`, `user.active`
Method	Action using that data	`user.deactivate()`
Class attr	Shared for all instances	`class User: role = "student"`

Guidelines:

- Keep attributes small and descriptive
- Use methods when behavior depends on the object’s state
- Return values from methods when you need reusable data (`describe()`)

Classes vs Simple Functions

```
Prefer simple functions when:
  - Logic is stateless
  - You just manipulate inputs/outputs quickly
  - The helper stands on its own (e.g., convert units)

Prefer classes when:
  - You repeatedly use the same bundle of data
  - Multiple behaviors belong to that bundle
  - You want to hide implementation details behind a clean API
```

You can mix both patterns: functions orchestrate flows, classes encapsulate objects.

Mini Task — Task Manager OOP

Inside `projects/task_manager/`:

1. Create `task.py` with a `Task` class containing `title`, `completed`, `priority="normal"`, and methods `mark_done()` plus `describe()` returning a formatted summary.
2. Create `main.py` that instantiates several tasks, stores them in a list, and prints each description.
3. Stretch goal: add a `TaskList` class with `add_task()`, `list_pending()` to practice classes working together.

This OOP layer will make it easier to structure file, database, and web projects in upcoming modules.

From Functions to Small Projects

Python Course · Module 5.5

ElnurBDa

Module 5.5 — Function-Driven Mini Projects

Goals:

- Use functions and modules to build a **slightly larger script**
- Practice the `main()` pattern with imports
- Prepare to add **files, logging, and error handling** in Module 6

Think: “one step up” from tiny examples toward real tools.

Review — Project Layout with Modules

From Module 5:

```
project/
├── main.py
├── utils.py
└── math_ops.py
```

Imports:

```
from utils import greet
from math_ops import multiply
```

In this module we'll build similar structures with **clear responsibilities**.

Designing a Small CLI Tool

We'll keep using `input()` for now, but we'll **separate logic**:

Example: simple unit converter project:

```
converter_project/
├── main.py
└── converter.py
```

`converter.py`:

```
KM_IN_MILE = 1.60934

def km_to_miles(km: float) -> float:
    return km / KM_IN_MILE

def celsius_to_fahrenheit(c: float) -> float:
    return c * 9 / 5 + 32
```

———————————— [finished] ————————————

`main.py`:

```
from converter import km_to_miles, celsius_to_fahrenheit

def main():
    choice = input("Convert (k)m or (t)emperature? ").strip().lower()
    if choice == "k":
        km = float(input("Kilometers: "))
        print("Miles:", km_to_miles(km))
    elif choice == "t":
        c = float(input("Celsius: "))
        print("Fahrenheit:", celsius_to_fahrenheit(c))
    else:
        print("Unknown option")

if __name__ == "__main__":
    main()
```

———————————— [finished with error] ————————————

```
Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presenterm6svXk1/snippet.py", line 1, in
<module>
```

Separating Pure Logic from I/O

Pure functions:

- Do not call `input()` or `print()`
- Only work with parameters and return values

This makes them:

- Easier to test
- Easier to reuse (CLI, web API, GUI)

In the previous example, all conversion logic is inside `converter.py`, while `main.py` only handles user interaction.

Module 6 will plug **file I/O** into a similar structure.

Mini Practice — Grade Utility

Design:

```
grades_project/
├── main.py
└── grades.py
```

`grades.py` should contain:

- `average(scores)`
- `letter_grade(score)` → `"A"`, `"B"`, ...
- `summarize(scores)` → dict with average, highest, lowest, pass_count

`main.py`:

- Reads a comma-separated list of numbers via `input()`
- Uses functions from `grades.py`
- Prints a short report

You will reuse this idea when reading grades from files in Module 6.

Introducing Simple Configuration

Before we start reading real config files, we can simulate configuration using a **separate module**:

```
settings_project/
├── config.py
└── main.py
```

config.py:

```
APP_NAME = "Demo App"
DEFAULT_LOG_LEVEL = "INFO"
MAX_ITEMS = 100
```

──────────────────── [finished] ────────────────────

main.py:

```
import config

def main():
    print("Starting", config.APP_NAME)
    print("Log level:", config.DEFAULT_LOG_LEVEL)
    print("Max items:", config.MAX_ITEMS)

if __name__ == "__main__":
    main()
```

──────────────── [finished with error] ─────────────────

```
Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermAnGYZc/snippet.py", line 1, in
<module>
    import config
ModuleNotFoundError: No module named 'config'
```

Later, Module 6/10 will replace some of these constants with **file- or env-based** configs.

Mini Project — Simple Menu-Driven App

Create `menu_project/`:

```
menu_project/
├── main.py
└── actions.py
```

`actions.py`:

- `def say_hello(name):` → prints greeting
- `def add_numbers(a, b):` → returns sum
- `def reverse_text(s):` → returns reversed string

`main.py`:

1. Shows a numbered menu: Hello, Add, Reverse, Quit.
2. Uses a loop + `if/elif` to call the right function in `actions.py`.
3. Keeps running until user chooses Quit.

This prepares you for more complex menus and CLIs in later modules.

Files & Error Handling

Python Course · Module 6

ElnurBDa

Module 6 — Files & Error Handling

In this module we learn how to **read and write files** and **handle errors gracefully**.

We will learn:

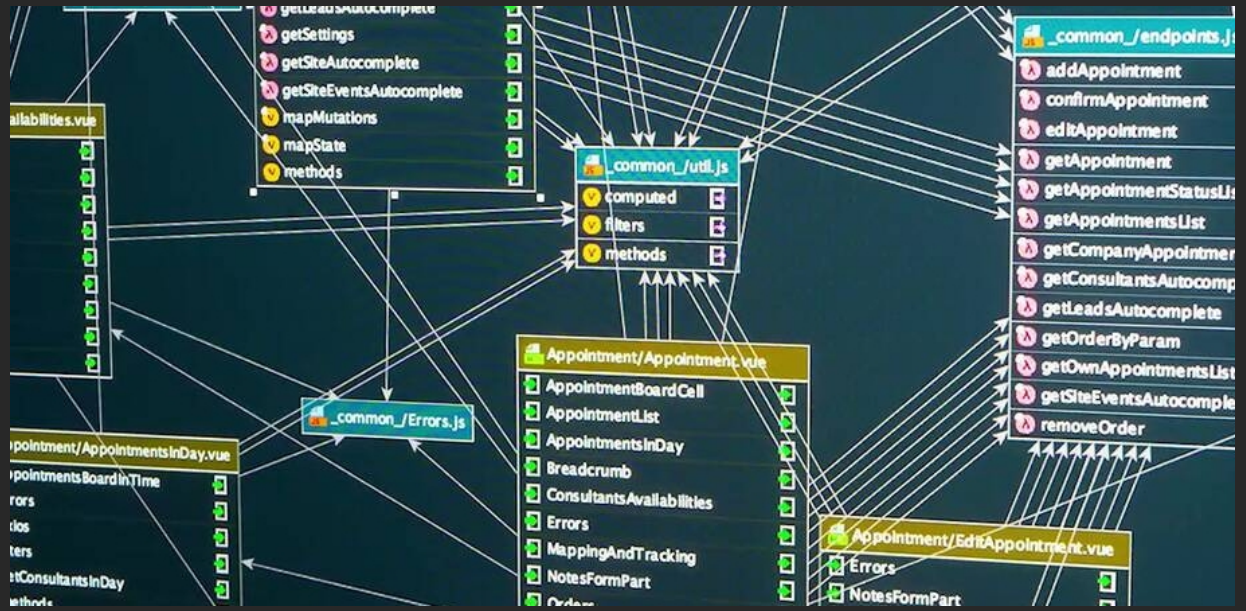
- Reading & writing files
- Handling errors with `try` / `except`
- Working with CSV files
- Best practices for error handling

Files & Persistent Storage

Programs normally store data in **RAM** (temporary).
Files store data on **disk** (persistent).

Why use files?

- Save logs
- Export reports
- Config files
- Simple data exchange



File Paths & Modes

To work with a file, Python needs:

- **Path** (where it is)
- **Mode** (what we want to do)

Common modes:

```
"r"   read (error if missing)
"w"   write, truncate file
"a"   append to end
"r+"  read & write
"rb"  read binary
"wb"  write binary
```

Always prefer **absolute paths** in bigger projects.

Opening Files Safely – with

Use with to open files – it auto-closes them.

```
# reading a text file
path = "notes.txt"

try:
    with open(path, "r", encoding="utf-8") as f:
        content = f.read()
        print("File content:")
        print(content)
except FileNotFoundError:
    print("File not found:", path)
```

———————— [finished] ————————

```
File content:
some test text is here
yay!
```

Benefits:

- No need to call close()
- Works well with exceptions

Reading Files — Variants

```
with open("notes.txt", "r", encoding="utf-8") as f:
    all_text = f.read()        # whole file
    print("LEN:", len(all_text))
```

———————————— [finished] ————————————

```
LEN: 28
```

```
with open("notes.txt", "r", encoding="utf-8") as f:
    first_line = f.readline()  # one line
    print("FIRST:", first_line)
```

———————————— [finished] ————————————

```
FIRST: some test text is here


```

```
with open("notes.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()      # list of lines
    print("COUNT:", len(lines))
```

———————————— [finished] ————————————

```
COUNT: 2
```

Choose based on file size and use-case.

———

Writing & Appending

```python
# overwrite or create
with open("log.txt", "w", encoding="utf-8") as f:
    f.write("First run\n")
    f.write("Program started successfully.\n")
```

———————————— [finished] ————————————

```python
# append
with open("log.txt", "a", encoding="utf-8") as f:
    f.write("Another run...\n")
```

———————————— [finished] ————————————

Notes:

- "w" **deletes** old content
- "a" keeps content and adds at the end

Working with CSV Files

CSV (Comma-Separated Values) is a simple text format for tables.

```
import csv

rows = [
    ["id", "username", "score"],
    [1, "admin", 99],
    [2, "guest", 42],
]

with open("users.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerows(rows)

with open("users.csv", "r", encoding="utf-8") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

```
['id', 'username', 'score']
['1', 'admin', '99']
['2', 'guest', '42']
```

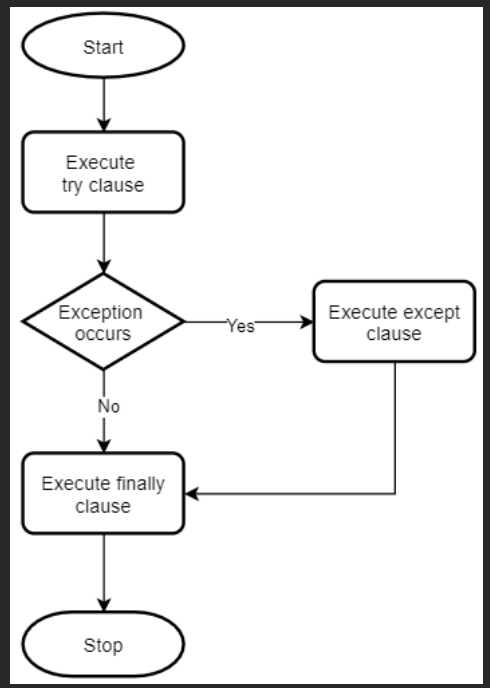
Exceptions – What & Why

An **exception** is a runtime error that stops normal execution.

Common sources:

- Invalid input
- Missing files
- Network errors
- Database problems

Goal: **fail safely**, show useful messages, keep program controllable.



Basic try / except

```python
# text = input("Enter a number: ")
text = "Salam"

try:
    n = int(text)
    print("Squared:", n * n)
except ValueError:
    print("That was not a valid integer!")
```

———————— [finished] ————————

```
That was not a valid integer!
```

Flow:

1. Run code in `try` block
2. If exception matches `except`, handle it
3. Program continues instead of crashing

Error Handling Patterns in Bigger Scripts

As programs grow, we want **clear places** where errors are handled.

Typical pattern:

```
def run():
    try:
        # 1) read config
        # 2) open files / database
        # 3) main logic
        ...
    except FileNotFoundError as e:
        print("Missing file:", e)
    except ValueError as e:
        print("Bad data:", e)
```

Benefits:

- All top-level errors are handled in **one place**
- Internal functions can either handle or **raise** exceptions

In real projects we often move these messages to **logging** instead of `print()`.

0_o

```python
# text = input("Enter a number: ")
text = "3"

try:
    n = int(text)
    print("Squared:", n * n)
except ValueError:
    print("That was not a valid integer!")
```

———————— [finished] ————————

```
Squared: 9
```

Catching Multiple Exceptions

```
def read_number_from_file(path):
    try:
        with open(path, "r", encoding="utf-8") as f:
            text = f.read().strip()
            return int(text)
    except FileNotFoundError:
        print("File does not exist:", path)
    except ValueError:
        print("File did not contain a valid number.")
```

You can also group them:

```
except (FileNotFoundError, PermissionError) as e:
    print("File access problem:", e)
```

---

else and finally

```python
path = "config.txt"

try:
    f = open(path, "r", encoding="utf-8")
except FileNotFoundError:
    print("Config missing")
else:
    print("Config loaded:", f.readline().strip())
    f.close()
finally:
    print("This always runs (cleanup, logs, etc.)")
```

———————————— [finished] ————————————

```
Config missing
This always runs (cleanup, logs, etc.)
```

- `else`: runs if **no** exception happened
- `finally`: runs **always**, used for cleanup

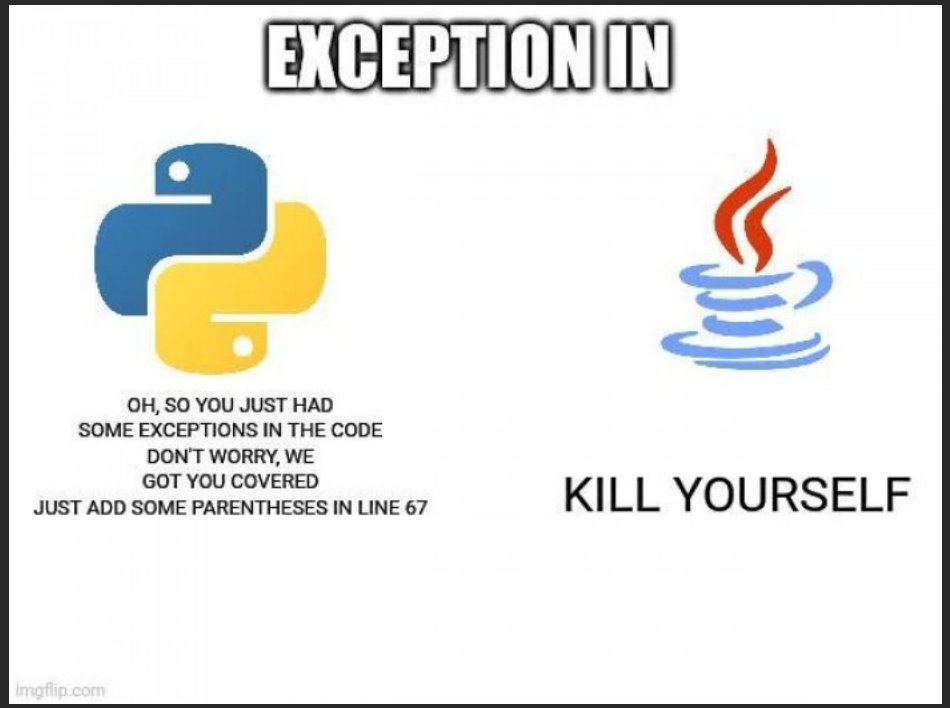
Mini Task

Create a **log tracker** program:

1. Ask user for their name and action
2. Write to activity.log with timestamp
3. Handle file errors gracefully
4. Read and display all logs
5. Add option to clear logs (with confirmation)

Bonus:

- Save logs as CSV with csv module
- Add logging levels (INFO, ERROR, WARNING)
- Filter logs by action or date



Files vs Databases

Python Course · Module 6.5

ElnurBDa

Module 6.5 — When to Use Files, When to Use Databases

Goals:

- Compare text/CSV files with relational databases
- Understand when simple files are enough
- See when you should move to **SQLite / SQL**

Think of this as a decision guide before you learn SQL syntax.

Recap — What Files Do Well

From Module 6, files are great for:

- Logs (`activity.log`)
- Configuration (`config.ini`, `.env`)
- Small exports (`report.txt`, `users.csv`)
- One-off data exchange between tools

Strengths:

- Very simple to inspect with any text editor
- Easy to copy, email, archive
- No extra server process required

Weaknesses:

- Harder to **search/filter** efficiently
- No built-in concept of relationships between rows
- Concurrency (many writers/readers) can be tricky

Recap — What Databases Do Well

From Module 7, relational databases (like SQLite) are great for:

- Structured data in **tables** with columns
- Fast querying with **indexes**
- Enforcing rules via constraints (`NOT NULL`, `UNIQUE`, `FOREIGN KEY`)
- Multiple readers (and some writers) at the same time

Strengths:

- Complex filters and aggregations with `WHERE`, `ORDER BY`, `JOIN`, etc.
- Central source of truth for your data
- Easier to evolve schema over time

Weaknesses:

- More concepts to learn (schema, types, constraints)
- Needs careful backups and migrations
- Not ideal for raw logs or one-off binary blobs

Scenario 1 – Log Tracker

Use **files** when:

- You’re writing simple append-only logs
- You mostly read them in full (or by lines)
- You don’t need complex queries

Possible upgrade:

- Move from plain text to **CSV** for slightly more structure
- Only move to a database if you need to filter heavily (e.g., by user/date)

Rule of thumb:

> If you only ever read all entries and scroll, a file is fine.

Scenario 2 – Score Tracker / Student Manager

For students, courses, scores:

- Many records (potentially hundreds or thousands)
- Need to search by email, course, status
- Might need to join with other tables later (attendance, tasks)

Files:

- Quick CSV exports for sharing or backup
- OK for tiny datasets with simple needs

Database:

- Better when you need to update scores often
- Safer with unique constraints on email
- Enables joins with other entities (e.g., students ↔ courses)

In this course, Module 7 moves these use-cases into SQLite.

Comparison Table – Files vs SQLite

Aspect	Files (txt/CSV)	SQLite (DB file)
Structure	Free-form / simple rows	Tables with columns & types
Validation	Manual in code	Constraints in schema
Queries	Manual loops/filtering	SQL (SELECT, WHERE, JOIN)
Concurrency	Basic, not coordinated	Better but still limited
Tools	Editors, Excel, scripts	sqlite3 CLI, DB browsers
Best for	Logs, configs, exports	Core application data

Takeaway:

- Start with files when prototyping
- Migrate to SQLite once queries or relations become complex

Migration Example — Users CSV → Users Table

Suppose you have users.csv:

```
id,username,email,score
1,admin,admin@example.com,99
2,guest,guest@example.com,42
```

You want to move to SQLite:

1. Design schema:

```sql
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    score INTEGER DEFAULT 0
);
```

2. Write a one-time Python script that:

- Reads users.csv with csv.reader
- Inserts each row into users using **parameterized** queries

After that, your main app can ignore the CSV and work only with SQLite.

Databases & SQL

Python Course · Module 7

ElnurBDa

Module 7 — Databases & SQL

In this module we connect **real-world programs** to **persistent, structured storage**.

We will learn:

- What databases are and when to use them
- Main database types
- How to use **SQLite** with Python
- Essential **SQL** syntax
- Building a simple database application

Files vs. Databases

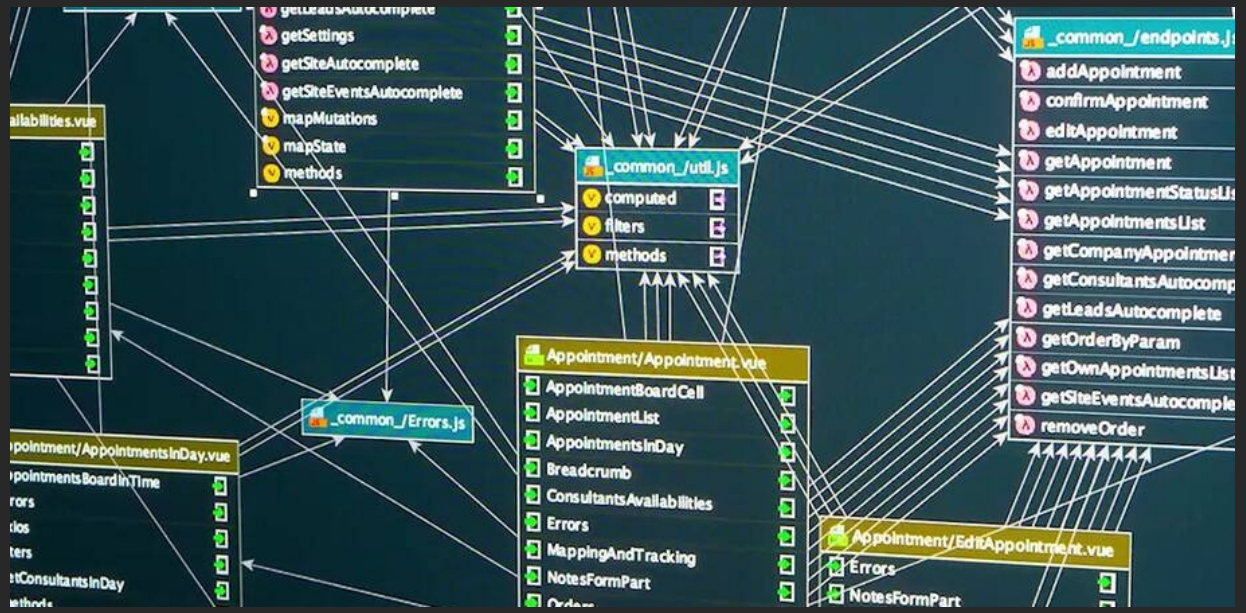
Programs normally store data in **RAM** (temporary).
Files and databases store data on **disk** (persistent).

Files are good for:

- Save logs
- Export reports
- Config files
- Simple data exchange

Databases are good for:

- Large data sets
- Structured relations
- Many users at once
- Fast searching / filtering



What is a Database?

A **database** is an organized collection of data.

Key features:

- Structured storage (tables / documents / graphs)
- Fast querying (indexes)
- Concurrency (many users at once)
- Security (permissions, access control)

Compared to raw files:

- Databases **understand** what the data means
- Easier to enforce rules (constraints)
- Optimized for large datasets

Main Database Types

SQL / Relational

- Tables: rows & columns
- Strong schema
- Uses SQL language
- Examples: PostgreSQL, MySQL, SQLite

Best for:

- Transactions
- Clear structure
- Reporting

NoSQL

- Flexible schema
- Designed for scale

Flavors:

- Key-value (Redis)
- Document (MongoDB)
- Graph (Neo4j)
- Wide-column (Cassandra)

SQLite – Lightweight SQL

SQLite is a **serverless** database engine.

Characteristics:

- Everything in **one file** (e.g. `app.db`)
- Excellent for desktop / small tools / testing
- Comes built-in with Python (`sqlite3` module)
- No separate install, no server process

Understanding Relational Databases — Tables, Rows & Columns

Think of a **table** like a **spreadsheet**:

id	username	email	score
1	alice	alice@example.com	100
2	bob	bob@example.com	85
3	charlie	charlie@example.com	92

- **Table** — the whole collection (like a sheet named "users")
- **Columns** — vertical categories (id, username, email, score)
- **Rows** — horizontal records (one per user)
- **Cell** — intersection of row & column (a single value)

Relational Databases — Why Relations?

Relational means tables are connected.

Example — two related tables:

USERS table:

```
id | username | score
---+----------+-------
1  | alice    | 100
2  | bob      | 85
3  | charlie  | 92
```

POSTS table:

```
id | user_id | title
---+---------+------------
1  | 1       | "Hello"
2  | 1       | "Hi again"
3  | 2       | "World"
```

The `user_id` in POSTS points to `id` in USERS.

Benefits:

- No duplicate data (alice's info stored once)
- Easy to update (change once, everywhere updates)
- Enforced connections (`FOREIGN KEY`)
- Prevent invalid references (no post from user_id 999)

Database Schema — Planning Your Data

Before creating tables, **plan your structure**:

1. **Identify entities** — What objects exist? (users, posts, comments)
2. **List attributes** — What info about each? (username, email, date)
3. **Choose types** — TEXT, INTEGER, REAL, etc.
4. **Define constraints** — Required? Unique? Default?
5. **Add relationships** — How do entities connect?

Example planning:

```
USERS:
- id (INTEGER, PRIMARY KEY, auto-increment)
- username (TEXT, NOT NULL, UNIQUE)
- email (TEXT, NOT NULL, UNIQUE)
- created_at (TEXT)

POSTS:
- id (INTEGER, PRIMARY KEY, auto-increment)
- user_id (INTEGER, FOREIGN KEY → users.id)
- title (TEXT, NOT NULL)
- content (TEXT)
- created_at (TEXT)
```

Connecting to SQLite

```python
import sqlite3

# creates file if it does not exist
conn = sqlite3.connect("app.db")

print("Connected:", conn)

conn.close()
```

———————— [finished] ————————

```
Connected: <sqlite3.Connection object at
0x7f4bcf356c50>
```

General pattern:

```python
conn = sqlite3.connect("app.db")
cur = conn.cursor()
# ... execute SQL here ...
conn.commit()
conn.close()
```

SQL Basics – Data Types

SQLite supports common data types:

```
INTEGER    whole numbers (1, -42, 1000)
REAL       floating point (3.14, 2.71)
TEXT       strings ("hello", "John")
BLOB       binary data (images, files)
NULL       no value (missing data)
```

Example:

```sql
CREATE TABLE products (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    price REAL,
    quantity INTEGER DEFAULT 0,
    description TEXT
);
```

SQL — CREATE TABLE

```sql
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    username TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    score INTEGER DEFAULT 0,
    created_at TEXT
);
```

Key constraints:

- `PRIMARY KEY` — unique identifier
- `NOT NULL` — value must be provided
- `UNIQUE` — no duplicate values
- `DEFAULT` — default value if not provided
- `AUTOINCREMENT` — auto-generate incrementing IDs

SQL — INSERT

Insert new rows into a table.

```sql
INSERT INTO users (username, email, score)
VALUES ('alice', 'alice@example.com', 100);
```

Insert multiple rows:

```sql
INSERT INTO users (username, email, score)
VALUES
    ('bob', 'bob@example.com', 85),
    ('charlie', 'charlie@example.com', 92),
    ('diana', 'diana@example.com', 78);
```

Always specify column names for clarity and safety.

SQL — SELECT

Query data from a table.

```sql
SELECT * FROM users;
```

Select specific columns:

```sql
SELECT username, score FROM users;
```

With WHERE condition:

```sql
SELECT * FROM users WHERE score > 80;
```

With multiple conditions:

```sql
SELECT * FROM users
WHERE score > 80 AND username LIKE 'a%';
```

SQL — SELECT Advanced

ORDER BY — sort results:

```
SELECT username, score FROM users ORDER BY score DESC;
```

LIMIT — restrict number of results:

```
SELECT * FROM users LIMIT 5;
```

OFFSET — pagination:

```
SELECT * FROM users LIMIT 5 OFFSET 10;
```

COUNT — count rows:

```
SELECT COUNT(*) FROM users;
```

Aggregate functions: SUM(), AVG(), MIN(), MAX()

SQL — UPDATE

Modify existing rows.

```sql
UPDATE users SET score = 150 WHERE username = 'alice';
```

Update multiple columns:

```sql
UPDATE users
SET score = 95, email = 'newemail@example.com'
WHERE username = 'bob';
```

Update with calculations:

```sql
UPDATE users SET score = score + 10 WHERE score < 50;
```

⚠️ **Always use WHERE clause** to avoid updating all rows!

SQL — DELETE

Remove rows from a table.

```
DELETE FROM users WHERE username = 'alice';
```

Delete multiple rows:

```
DELETE FROM users WHERE score < 10;
```

Delete all rows (keep table structure):

```
DELETE FROM users;
```

⚠️ **Always use WHERE clause** — without it, all rows are deleted!

SQL — Relationships & Foreign Keys

Create related tables:

```
CREATE TABLE posts (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    title TEXT NOT NULL,
    content TEXT,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

FOREIGN KEY ensures:

- Only valid user IDs can be referenced
- Data integrity
- Prevents orphaned posts

SQL — JOIN

Combine data from multiple tables.

```sql
SELECT users.username, posts.title
FROM posts
JOIN users ON posts.user_id = users.id;
```

Different JOIN types:

- `INNER JOIN` — matching rows only
- `LEFT JOIN` — all from left table
- `RIGHT JOIN` — all from right table
- `FULL OUTER JOIN` — all from both tables

Example with filtering:

```sql
SELECT users.username, COUNT(posts.id) as post_count
FROM users
LEFT JOIN posts ON users.id = posts.user_id
GROUP BY users.id
HAVING COUNT(posts.id) > 0;
```

SQLite CLI

SQLite has a **command-line tool** for quick testing.

Open a database:

```
sqlite3 app.db
```

You'll see:

```
SQLite version 3.x.x
sqlite>
```

Common commands:

```
.tables              -- list tables
.schema              -- show CREATE statements
.exit                -- quit
.mode column         -- prettier output
.headers on          -- show column names
```

SQLite CLI — Running SQL

```
sqlite> CREATE TABLE users (
...>     id INTEGER PRIMARY KEY AUTOINCREMENT,
...>     username TEXT NOT NULL UNIQUE,
...>     score INTEGER DEFAULT 0
...> );

sqlite> INSERT INTO users (username, score) VALUES ('alice', 100);
sqlite> INSERT INTO users (username, score) VALUES ('bob', 85);

sqlite> SELECT * FROM users;
id|username|score
1|alice|100
2|bob|85

sqlite> SELECT * FROM users WHERE score > 80 ORDER BY score DESC;
id|username|score
1|alice|100
2|bob|85

sqlite> UPDATE users SET score = 95 WHERE username = 'bob';

sqlite> .exit
```

Tip: Press ↑ to recall previous commands!

Using SQL in Python

```python
import sqlite3

conn = sqlite3.connect("app.db")
cur = conn.cursor()

sql = (
    "CREATE TABLE IF NOT EXISTS users ("
    " id INTEGER PRIMARY KEY AUTOINCREMENT,"
    " username TEXT NOT NULL UNIQUE,"
    " score INTEGER DEFAULT 0"
    ")"
)
cur.execute(sql)

conn.commit()
conn.close()
print("Table created.")
```

———————————— [finished] ————————————

```
Table created.
```

Notes:

- IF NOT EXISTS avoids errors
- PRIMARY KEY gives unique id

CRUD — Insert & Select

```python
import sqlite3

conn = sqlite3.connect("app.db")
cur = conn.cursor()

cur.execute(
    "INSERT INTO users (username, score) VALUES (?, ?)",
    ("admin", 100),
)

cur.execute("SELECT id, username, score FROM users")
rows = cur.fetchall()

for row in rows:
    print(row)

conn.commit()
conn.close()
```

———————————— [finished with error] ————————————

```
Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermnBl8gK/snippet.py", line 6, in
<module>
    cur.execute(
    ~~~~~~~~~~~^
        "INSERT INTO users (username, score) VALUES (?, ?)",
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        ("admin", 100),
        ^^^^^^^^^^^^^^^
    )
    ^
sqlite3.IntegrityError: UNIQUE constraint failed: users.username
```

? placeholders prevent SQL injection.

CRUD — Update & Delete

```python
import sqlite3

conn = sqlite3.connect("app.db")
cur = conn.cursor()

cur.execute(
    "UPDATE users SET score = ? WHERE username = ?",
    (150, "admin"),
)

cur.execute(
    "DELETE FROM users WHERE username = ?",
    ("guest",),
)

conn.commit()
conn.close()
```

```
──────────────────── [finished] ────────────────────
```

Always `commit()` after changes.

Putting It Together

```python
import sqlite3

def get_conn():
    return sqlite3.connect("app.db")

def add_user(username, score):
    with get_conn() as conn:
        cur = conn.cursor()
        cur.execute(
            "INSERT INTO users (username, score) VALUES (?, ?)",
            (username, score),
        )

def list_users():
    with get_conn() as conn:
        cur = conn.cursor()
        cur.execute("SELECT id, username, score FROM users")
        return cur.fetchall()
```

Pattern:

- Small helper functions
- Use `with` for connection lifetime

Mini Task

Create a mini **score tracker**:

1. Create SQLite DB & users table
2. Ask user for username and score
3. Insert using **parameterized** query
4. Select and print all users
5. Add option to **update** score

Bonus:

- Create a posts table
- Link posts to users with FOREIGN KEY
- Show user + their posts

From Database to API

Python Course · Module 7.5

ElnurBDa

Module 7.5 — Turning Queries into Endpoints

Goals:

- Map your existing **CRUD functions** to HTTP verbs
- Understand how one database query becomes an **API endpoint**
- Prepare for the full To-Do / Student Manager apps in Module 9

Think of this as “wrapping SQL in HTTP”.

Recap — CRUD with SQLite in Python

From Module 7, basic pattern:

```
import sqlite3

def get_conn():
    return sqlite3.connect("app.db")

def add_user(username, score):
    with get_conn() as conn:
        cur = conn.cursor()
        cur.execute(
            "INSERT INTO users (username, score) VALUES (?, ?)",
            (username, score),
        )
```

———————————— [finished] ————————————

We already have **functions** that implement:

- Create (INSERT)
- Read (SELECT)
- Update (UPDATE)
- Delete (DELETE)

Now we'll think about how to **expose** them over HTTP.

CRUD → HTTP Verbs

Typical mapping:

Action	SQL	HTTP verb	Example path
Create	INSERT	POST	/api/users
Read many	SELECT	GET	/api/users
Read one	SELECT	GET	/api/users/<id>
Update	UPDATE	PUT/PATCH	/api/users/<id>
Delete	DELETE	DELETE	/api/users/<id>

Key idea:

- Your existing Python DB functions become the **implementation** behind these endpoints.
- A web framework (like Flask in Module 9) will handle parsing HTTP and calling them.

This module focuses on the **conceptual mapping**.

Designing a Tiny Users API (Concept)

Assume we have these DB helpers:

```
def list_users():
    ...

def get_user(user_id):
    ...

def create_user(username, email):
    ...
```

We could imagine HTTP routes:

- `GET /api/users` → call `list_users()`
- `GET /api/users/<id>` → call `get_user(id)`
- `POST /api/users` → read JSON body, call `create_user(...)`

Later, Module 9 will show actual Flask code that wires this together.

Example — Pseudocode for an Endpoint

Pseudocode, not real Flask code yet:

```
def handle_get_users_request():
    # 1. Read query params (e.g., ?min_score=80)
    # 2. Call list_users(min_score)
    # 3. Convert rows to JSON-serializable dicts
    # 4. Return JSON + status code 200
    ...
```

The main job of an **endpoint**:

1. Validate and parse **HTTP input** (URL, query, body)
2. Call core logic / DB helpers (pure Python)
3. Format a proper **HTTP response** (status code + headers + body)

Your SQL logic from Module 7 lives entirely in step 2.

Mini Design — Tasks Table to Tasks API

Suppose we have a `tasks` table:

```
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    completed INTEGER NOT NULL DEFAULT 0
);
```

DB helpers:

```
def create_task(title):
    ...

def list_tasks(completed=None):
    ...

def mark_task_done(task_id):
    ...
```

Matching endpoints:

- `POST /api/tasks` → `create_task(title)`
- `GET /api/tasks?completed=true` → `list_tasks(completed=True)`
- `PATCH /api/tasks/<id>` or `PUT` → `mark_task_done(id)`

You'll implement these with Flask in Module 9.

Practice — Map Your Queries to Routes

Take your own SQLite mini-project from Module 7:

- Score tracker
- Student list
- Any custom table you built

For each query / function, design:

1. An HTTP method (GET, POST, PATCH, DELETE)
2. A URL path (e.g., /api/students, /api/students/<id>)
3. What parameters come from:
   - path (<id>)
   - query (?course=Python)
   - JSON body ({"name": "...", "email": "..."})

Write this as a small table in your notes.

Thinking About Status Codes

Common choices:

- 200 OK – successful read/update
- 201 Created – new resource created (e.g., new user/task)
- 400 Bad Request – invalid input (fails validation)
- 404 Not Found – resource with that ID does not exist
- 500 Internal Server Error – unexpected server crash

For each planned endpoint, decide:

- What should a **successful** response look like?
- What errors can happen (invalid data, missing row)?
- Which status code makes the most sense in each case?

This will plug directly into Flask route handlers later.

Preview — Flask Style

From Module 9, an actual Flask handler looks like:

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.get("/api/tasks")
def list_tasks_route():
    completed = request.args.get("completed")
    # call list_tasks(completed=...)
    # return jsonify([...]), 200
```

Notice:

- URL and HTTP method are declared in the decorator
- Query params come from `request.args`
- Response is produced with `jsonify` and a status code

Your job now is just to design **what** routes you want and which DB functions they call.

Mini Project – API Blueprint for Your App

Pick one project:

- Score tracker
- Student Manager
- To-Do list with SQLite backend (if you've built it)

Create an **API blueprint** document (markdown / text) that lists:

1. Tables involved and their key columns.
2. All planned endpoints with:
   - HTTP method + path
   - Short description
   - Example request (URL + JSON body if relevant)
   - Example JSON response
3. Which existing Python function(s) each endpoint will call.

You can drop this blueprint straight into Module 9 when you actually build the Flask app.

Web Basics

Python Course · Module 8

ElnurBDa

Module 8 — Web Basics

We connect Python skills to the web stack:

We will go in three stages:

1. **Web pages** — HTML, CSS, JavaScript, and how the browser talks to a server
2. **HTTP & APIs** — requests, responses, status codes, and JSON
3. **Python as a web client/server** — using requests and serving simple pages

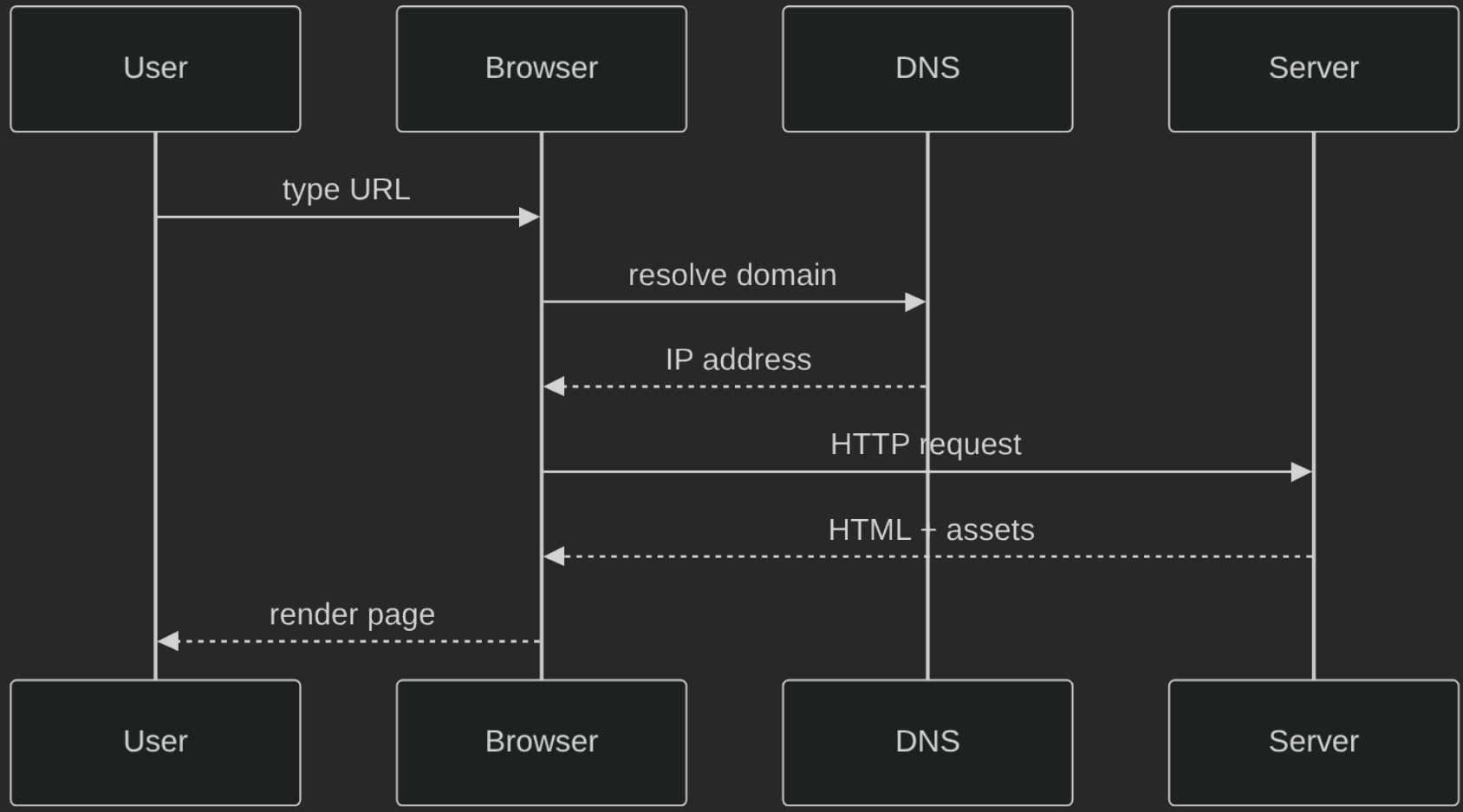
Keep this question in mind:

> “What exactly is traveling over the network right now — HTML or JSON?”

How the Web Works

Step-by-step workflow from browser to server:

1. User enters https://example.com
2. Browser resolves domain → IP
3. Browser sends HTTP request
4. Server responds with HTML, CSS, JS
5. Browser renders page



Section 1 — Web Pages (HTML, CSS, JS)

In this first part we focus on **how a page is built and served**. Do not worry about APIs or JSON yet — just think in terms of:

- HTML = structure & content
- CSS = visual style
- JavaScript = interactivity in the browser

Later sections will reuse this when we load data from APIs instead of hard-coding it.

HTML — Building Blocks

HTML describes structure/content of web pages (like blueprints of a house).

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Hello Web</title>
  </head>
  <body>
    <h1>Welcome!</h1>
    <p>This is my first web page.</p>
    <a href="https://python.org">Python.org</a>
  </body>
</html>
```

- Tags usually come in pairs: `<tag>content</tag>`
- Attributes add metadata: `<a href="https://python.org">`
- Browser builds the **DOM tree** from these nested tags

HTML Storyboard: Code → Server → Browser

1. **Write HTML** (index.html)
2. **Serve it** (python3 -m http.server 8000)
3. **View in browser** (http://localhost:8000)

```
[index.html] --(http.server)--> [localhost:8000] --(render)--> [Page]
```

Focus questions:

- Do tags nest correctly? (<main> contains <section>)
- Did you include alt text for images?
- Can you describe each section aloud?

HTML Document Anatomy

Section	Purpose
`<!DOCTYPE>`	Tells browser to use modern HTML5 rules
`<html>`	Root element wrapping everything
`<head>`	Metadata, title, linked CSS/JS
`<body>`	Visible content: text, images, forms

Example with more tags:

```
<body>
  <header><h1>Cyber Café</h1></header>
  <main>
    <section>
      <h2>Menu</h2>
      <ul>
        <li><strong>Espresso</strong> — 3 AZN</li>
        <li><strong>Tea</strong> — 2 AZN</li>
      </ul>
    </section>
    <img src="assets/cafe.jpg" alt="Cafe interior" />
  </main>
  <footer>&copy; 2025 Cyber Café</footer>
</body>
```

Core HTML Tags & Patterns

Tag	What it does	Example snippet
`<h1>`	Page title	`<h1>News</h1>`
`<p>`	Paragraph	`<p>Breaking updates...</p>`
`<a>`	Link to another page/file	`<a href="/about.html">About</a>`
`<img>`	Display image	`<img src="logo.png" alt="Logo" />`
`<ul>/<ol>`	Lists (unordered/ordered)	`<ul><li>Item</li></ul>`
`<form>`	User input	`<form><input name="email" /></form>`
`<div>`/`<span>`	Layout/grouping	`<div class="card">...</div>`

Tip: Start simple, add CSS later to style it.

Styling with CSS

CSS (Cascading Style Sheets) controls presentation.

Ways to include CSS:

- Inline: `<p style="color:red">`
- `<style>` block in `<head>`
- External file: `<link rel="stylesheet" href="styles.css" />`

Example:

```html
<head>
  <link rel="stylesheet" href="assets/styles.css" />
</head>
<body>
  <h1 class="hero">Welcome</h1>
</body>
```

```css
.hero {
  color: #ff9900;
  text-align: center;
  border-bottom: 2px solid #444;
}
```

Guideline: keep structure (HTML) separate from style (CSS) for clarity.

JavaScript in HTML

JavaScript brings interactivity.

Common patterns:

- Inline handlers: `<button onclick="alert('Hi')">`
- `<script>` tag inside HTML file
- External file: `<script src="app.js"></script>`

Example:

```
<body>
  <p id="greeting">Loading...</p>
  <button id="btn">Say hi</button>

  <script>
    const message = "Hello from JS!";
    document.getElementById("btn").addEventListener("click", () => {
      document.getElementById("greeting").textContent = message;
    });
  </script>
</body>
```

Place scripts near the end of `<body>` so HTML loads before JS runs.

Serving index.html Quickly

Browsers look for index.html by default.

```
cd website_project
python3 -m http.server 8000
```

- Place index.html alongside the command
- Visit http://localhost:8000 to see it rendered
- Great for checking static prototypes before moving to frameworks

Challenge: create index.html with header, paragraph, list, and image, then serve it locally.

Section 2 — HTTP & APIs

Now that you know what a **page** looks like, we zoom in on the **conversation** between browser and server.

Questions to keep in mind:

- Who is the **client** in this example? Who is the **server**?
- What is the **URL**, what is the **method** (GET/POST/...), and what is inside the **body** (if any)?
- Is the response returning **HTML** or **JSON**?

Once this feels comfortable, we will replace “browser” with **Python code** using `requests`.

HTTP — The Conversation Rules

HTTP (HyperText Transfer Protocol) defines how clients & servers talk — like a script both sides follow.

Structure:

```
<METHOD> <PATH> HTTP/1.1
Header: Value
Header: Value

<optional body>
```

Request example:

```
GET /api/users HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: application/json
```

Response example:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"status": "ok", "count": 2}
```

Common verbs:

- GET (read), POST (create), PUT/PATCH (update), DELETE (remove)

HTTP Anatomy — Request vs Response

Request

```
POST /login HTTP/1.1
Host: mysite.com
Content-Type: application/json
Authorization: Bearer <token>

{"email": "user@example.com", "password": "***"}
```

- Method + path
- Headers describe metadata (auth, format)
- Body carries data (optional)

Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Set-Cookie: session=abc123

{"status": "ok", "message": "Welcome"}
```

- Status line communicates result
- Headers for caching, cookies, etc.
- Body returns JSON/HTML/binary

Status Codes

Code	Meaning	Use case
200	OK	Successful request
201	Created	Resource created (POST)
301	Moved Permanently	Redirect
400	Bad Request	Invalid client data
401	Unauthorized	Missing/invalid auth
404	Not Found	Resource missing
500	Internal Server Error	Server bug/problem

Fun reference with cat photos: https://http.cat

APIs — Digital Waiters

API = Application Programming Interface.

- Think of a restaurant: you (client) tell the waiter (API) what you want, they talk to the kitchen (server), and bring back the dish (data).
- HTTP APIs expose data/actions via URLs, verbs, and JSON payloads.
- Good APIs document their menu (endpoints, parameters, status codes).

Example: Tasks API served from http://localhost:3000

Endpoint	Method	Description
/api/tasks?completed=true	GET	List tasks filtered by completion flag
/api/tasks	POST	Create a new task with title/description flags
/api/tasks/{id}	GET	Fetch one task by its numeric identifier
/api/tasks/{id}	PUT	Update title, description, or completed status
/api/tasks/{id}	DELETE	Remove the task forever
/api/health	GET	Quick health check for the service

Menu analogy:

- Endpoint = dish name
- Query params/body = customization
- Response JSON = meal you receive

JSON — Data Containers

JSON (JavaScript Object Notation) represents structured data with nested objects/lists.

```
{
  "id": 42,
  "username": "admin",
  "scores": [99, 88, 91],
  "profile": {
    "city": "Baku",
    "active": true
  }
}
```

Analogy: dictionaries + lists combined.

Rules of thumb:

- Keys in double quotes, values can be string/number/bool/null/object/array
- Order usually not important, whitespace ignored

Another sample (array of objects):

```
[
  {"task": "learn HTML", "done": true},
  {"task": "build API client", "done": false}
]
```

Analogy: shipping boxes inside a truck — arrays hold boxes, each box (object) has labeled compartments (keys).

From HTTP & JSON to Python Code

Let’s connect the dots:

1. Browser or client sends an **HTTP request** (method + path + headers + body).
2. Server sends back an **HTTP response** with a **status code** and **body**.
3. If `Content-Type: application/json`, the body contains **JSON**, which feels like Python `dict` + `list`.
4. Our Python script with `requests` plays the role of the **client** instead of the browser.

So when you see `response.json()` later, imagine:

> “Take the JSON body from the HTTP response and turn it into Python dictionaries/lists I can work with.”

Inspecting APIs

Tools:

- Browser dev tools → Network tab
- `curl` in terminal:

```
curl -G "http://localhost:3000/api/tasks" \
  --data-urlencode "completed=true"
```

- Postman, Bruno, or Insomnia for graphical testing

Always check: URL, method, headers, body, response code, payload. Tasks API also requires you to pass `completed` as a query parameter when listing.

Python requests Basics

```python
import requests

url = "http://localhost:3000/api/tasks"
params = {"completed": "false"}  # API requires this flag
response = requests.get(url, params=params, timeout=5)

print("Status:", response.status_code)
response.raise_for_status()
tasks = response.json()
print("First task title:", tasks[0]["title"] if tasks else "No tasks yet")
```

snippet +exec is disabled, run with -x to enable

Notes:

- `timeout` prevents hanging forever (network problems)
- `response.status_code` shows the HTTP status (200, 404, 500, …)
- `response.raise_for_status()` throws an exception for 4xx/5xx errors
- `response.text` is the raw string body, `response.json()` parses JSON into Python types

---

Checkpoint — Your First API Call

1. Activate your venv
2. `pip install requests`
3. Run:

```python
import requests

BASE_URL = "http://localhost:3000/api/tasks"
data = requests.get(
    BASE_URL,
    params={"completed": "true"},
    timeout=5,
).json()

print("Total completed tasks:", len(data))
print("Sample record:", data[0] if data else "None")
```

4. Screenshot output for homework

Questions to consider:

- What happens if you forget `.json()`?
- How would you handle an empty list or missing key?

Section 3 — Python as Web Client & Simple Server

In this final part we:

1. Use Python + `requests` as an **HTTP client** talking to the Tasks API.
2. Use `http.server` (and a tiny custom server) as a **simple HTTP server** to serve static HTML.

Key mental model:

- Browser and Python scripts are just **different HTTP clients**.
- Servers (your simple Python server, or the Tasks API on port 3000) all speak the same HTTP "language".

When in doubt, go back to the HTTP request/response examples and map each line to what `requests` or the browser is doing.

Installing requests Inside a venv

```
python3 -m venv .venv
source .venv/bin/activate  # Windows: .venv\Scripts\activate
pip install --upgrade pip
pip install requests
pip freeze
```

- Always isolate dependencies per project
- `pip freeze` helps document versions (`requirements.txt`)
- Deactivate with `deactivate` when done

Sending Data with requests

```python
import requests

payload = {
    "title": "Draft Module 8 slides",
    "description": "Cover Tasks API examples",
    "completed": False,
}

resp = requests.post(
    "http://localhost:3000/api/tasks",
    json=payload,
    timeout=5,
)

resp.raise_for_status()
created = resp.json()
print("New task id:", created["id"])
print("Completed flag:", created["completed"])
```

snippet +exec is disabled, run with -x to enable

`json=...` auto converts dict to JSON & sets `Content-Type: application/json`.

Serving Static HTML with Python

Quick demo using built-in http.server.

```
python3 -m http.server 8000
```

- Serves current directory
- Visit http://localhost:8000 to view files
- Great for testing HTML/CSS locally

Stop with Ctrl+C.

Custom Minimal Server

Serve a specific HTML file with Python.

```python
from http.server import SimpleHTTPRequestHandler, HTTPServer
from pathlib import Path

PORT = 8080
ROOT = Path("website")

class StaticHandler(SimpleHTTPRequestHandler):
    def translate_path(self, path):
        # Map requests into our ROOT directory
        return str(ROOT / path.lstrip("/"))

if __name__ == "__main__":
    ROOT.mkdir(exist_ok=True)
    httpd = HTTPServer(("0.0.0.0", PORT), StaticHandler)
    print(f"Serving on http://localhost:{PORT}")
    httpd.serve_forever()
```

snippet +exec is disabled, run with -x to enable

Place `index.html` inside `website/` and run script.

Explanation:

- `HTTPServer` listens on port 8080 and waits for requests
- `SimpleHTTPRequestHandler` knows how to serve files; we override `translate_path` to force it into `website/`
- `ROOT.mkdir(exist_ok=True)` ensures the folder exists before serving
- Visit http://localhost:8080/index.html to test

Project structure reference:

```
website/
├── index.html
├── about.html
└── assets/
    └── styles.css
server.py
```

Run `python server.py` from the folder containing `website/`.

Mini Project

Build a simple task dashboard backed by the local API:

1. Create `index.html` that renders a table of tasks from `data.json`.
2. Write `fetch_tasks.py` that calls `http://localhost:3000/api/tasks?completed=false` and saves the JSON payload.
3. Add a second button on the page that loads tasks marked as completed (read from another JSON file or switch endpoints dynamically).
4. Serve the folder via `python -m http.server` and demo reloading after rerunning the fetch script.

Web Apps in Practice

Python Course · Module 9

ElnurBDa

Module 9 — Web App Architecture & To-Do App

In this module we connect everything together in a **simple web application**:

- Basic web application architecture
- Frontend vs backend vs database
- Building a minimal backend for a To-Do app
- Building a minimal frontend for it
- Extra mini-project for practice

Big Picture — Web Application Architecture

Modern web apps are usually split into three main parts:

- **Frontend** — code running in the browser (HTML/CSS/JS)
- **Backend** — server code handling requests and business logic
- **Database** — persistent storage for data

The browser never talks to the database directly — it always goes through the backend.

Roles of Frontend, Backend, Database

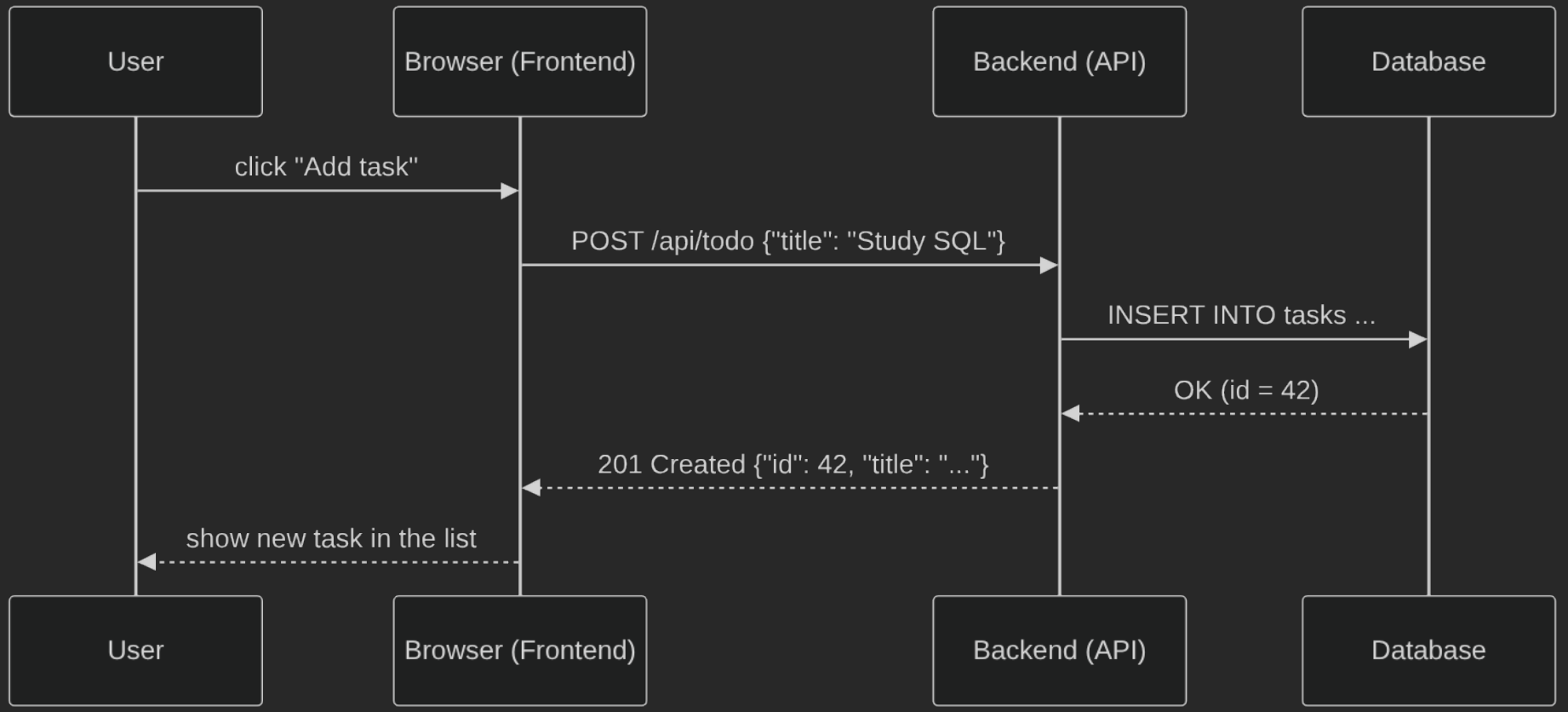
Layer	Main role	Technologies (examples)
Frontend	UI, layout, user interactions	HTML, CSS, JavaScript
Backend	Logic, validation, security, APIs	Python, Node, Java, Go, etc.
Database	Store and query structured data	SQLite, PostgreSQL, MySQL, etc.

Analogy:

- Frontend = restaurant dining area & menu
- Backend = kitchen + waiters
- Database = pantry where ingredients are stored

Request / Response Flow

User action → network request → backend → database → response → UI update.



Key idea: **HTTP** is the language between frontend and backend.

Designing a Simple To-Do Backend

We will build a tiny **JSON API** in Python for a To-Do app.

Features:

- List all tasks
- Create new task
- Mark task as completed

Simplest storage option:

- Start with an **in-memory list** (lost on restart but easy to understand)
- Later you can swap it with SQLite (from Module 7)

To-Do Backend — Minimal API (Flask)

todo_backend.py

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

tasks = []  # in-memory list of dicts
next_id = 1

@app.get("/api/todo")
def list_tasks():
    return jsonify(tasks)

@app.post("/api/todo")
def create_task():
    global next_id
    data = request.get_json() or {}
    title = data.get("title", "").strip()
    if not title:
        return {"error": "title is required"}, 400

    task = {"id": next_id, "title": title, "completed": False}
    next_id += 1
    tasks.append(task)
    return task, 201

if __name__ == "__main__":
    app.run(debug=True)
```

snippet +exec is disabled, run with -x to enable

Run with:

```
pip install flask
python todo_backend.py
```

Testing the Backend with curl

```
# list tasks
curl http://localhost:5000/api/todo

# create a new task
curl -X POST http://localhost:5000/api/todo \
  -H "Content-Type: application/json" \
  -d '{"title": "Finish Module 9 slides"}'
```

Check:

- HTTP method (GET, POST)
- URL path (/api/todo)
- Request body and headers
- Response JSON and status code

Optional: Persist To-Dos with SQLite

You can reuse **Module 7** knowledge to save tasks in app.db.

Sketch:

```
CREATE TABLE IF NOT EXISTS tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    completed INTEGER NOT NULL DEFAULT 0
);
```

```
import sqlite3

def get_conn():
    return sqlite3.connect("app.db")
```

Then inside your Flask handlers, instead of using a global tasks list, execute INSERT, SELECT, and UPDATE queries.

Building the Frontend — HTML Skeleton

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Mini To-Do App</title>
    <style>
      body { font-family: system-ui, sans-serif; margin: 2rem; background: #1d2021; color: #ebdbb2; }
      h1 { color: #fabd2f; }
      input, button { padding: 0.4rem 0.6rem; margin-right: 0.4rem; }
      ul { list-style: none; padding: 0; }
      li { padding: 0.25rem 0; }
    </style>
  </head>
  <body>
    <h1>Mini To-Do App</h1>
    <input id="title" placeholder="New task..." />
    <button id="add-btn">Add</button>
    <ul id="list"></ul>

    <script src="app.js"></script>
  </body>
  </html>
```

Serve this folder with:

```
python3 -m http.server 8000
```

Frontend Logic — Fetching Tasks

app.js

```
const API_URL = "http://localhost:5000/api/todo";

async function loadTasks() {
  const res = await fetch(API_URL);
  const data = await res.json();

  const list = document.getElementById("list");
  list.innerHTML = "";
  for (const task of data) {
    const li = document.createElement("li");
    li.textContent = `${task.id}. ${task.title}`;
    list.appendChild(li);
  }
}

loadTasks();
```

Open http://localhost:8000 in a browser to see the list (empty at first).

Frontend Logic – Creating Tasks from UI

Extend app.js:

```
const input = document.getElementById("title");
const button = document.getElementById("add-btn");

button.addEventListener("click", async () => {
  const title = input.value.trim();
  if (!title) return;

  await fetch(API_URL, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ title }),
  });

  input.value = "";
  await loadTasks();
});
```

Flow:

- User types a title and clicks **Add**
- Browser sends POST request to backend
- Backend adds task and returns JSON
- Frontend reloads the list using loadTasks()

Bringing It All Together

End-to-end:

1. Start backend:
   - `python todo_backend.py` → serves `http://localhost:5000/api/todo`
2. Start static frontend server in same folder as `index.html`:
   - `python3 -m http.server 8000`
3. Visit `http://localhost:8000` in browser:
   - Add tasks from UI
   - Confirm requests in **Network** tab of DevTools
4. Observe:
   - Frontend ↔ Backend via HTTP
   - Backend may use in-memory list or SQLite

You just built a tiny full-stack application.

Mini Project — Student Manager

Build a mini student management tool that reuses the same architecture:

1. **Backend API**:
   - Endpoints: `GET /api/students`, `POST /api/students`, `PATCH /api/students/<id>`
   - Each student record includes `id`, `name`, `email`, `course`, and `status` (active/inactive)
2. **Database (recommended)**:
   - SQLite table `students` with columns `(id INTEGER PRIMARY KEY, name TEXT, email TEXT UNIQUE, course TEXT, status TEXT)`
   - Seed 3 demo rows for quick testing
3. **Frontend**:
   - Table showing all students (name, course, status badge)
   - Form to add a new student (name, email, course)
   - Button to toggle status between Active / Inactive (calls PATCH)

Stretch ideas:

- Filter students by course or status
- Add simple validation and inline error messages
- Export the current list to JSON for download

Secure Python Basics

Python Course · Module 10

ElnurBDa

Module 10 — Security & Privacy Basics

In this module we connect your Python, database, and web skills to **security**:

- Trust boundaries & input validation
- Common pitfalls in CLI, database, and web apps
- SQL injection and **parameterized queries**
- Handling secrets and sensitive data (passwords, tokens, logs)

Goal: build a **security mindset** for all the code you write next.

Why Security Matters in Everyday Code

Even “small” scripts can:

- Delete or leak files
- Corrupt databases
- Expose personal data (names, emails, grades)

Realistic risks:

- Users type unexpected / malicious input
- Someone calls your API with weird parameters
- Logs or config files accidentally contain passwords or tokens

Security is not just about hacking tools — it’s about **writing code that behaves safely** under bad input.

Trust Boundaries — Who Do You Trust?

Every program has **trust boundaries**:

- Data you fully control (constants, internal helpers)
- Data from **users** (CLI `input()`, HTML forms, API requests)
- Data from **external systems** (files, databases, 3rd-party APIs)

Rule of thumb:

> Anything that crosses a boundary into your program is **untrusted** until checked.

We'll see how this affects:

- Input validation
- SQL queries
- Logging and error messages

Validating User Input (CLI)

Example: simple CLI for age input.

Unsafe style (no checks, will crash on bad data):

```
age = int(input("Enter your age: "))
print("Next year:", age + 1)
```

Safer style (validate + handle errors):

```
def ask_age():
    text = input("Enter your age: ").strip()
    if not text.isdigit():
        print("Please enter only digits.")
        return None

    age = int(text)
    if age <= 0 or age > 130:
        print("Please enter a realistic age.")
        return None

    return age

age = ask_age()
if age is not None:
    print("Next year you will be", age + 1)
```

———————— [finished with error] ————————

```
Enter your age: Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermYqlMmk/snippet.py", line 14, in
<module>
    age = ask_age()
  File "/tmp/nix-shell-61594-0/.presentermYqlMmk/snippet.py", line 2, in
ask_age
    text = input("Enter your age: ").strip()
           ~~~~~^^^^^^^^^^^^^^^^^^^^
EOFError: EOF when reading a line
```

Pattern:

- **Validate** format and range
- **Fail gracefully** with a clear message

Validating Web/API Input (Concept)

In web apps (like your Module 9 To-Do or Student Manager):

- Every field from a form or JSON body is **untrusted**.
- You must check: **type**, **length**, **allowed characters**, **business rules**.

Examples of validation rules:

- `username`: 3–32 chars, letters/numbers/underscores only
- `email`: contains `@` and a domain (use a simple regex or library)
- `title` (task): non-empty, max 200 chars

Idea: keep validation **close to the boundary** (in the route handler / controller) so the rest of your code can assume data is clean.

SQL Injection — The Problem

From Module 7 you know how to build SQL queries. Now we look at a **classic attack**: SQL injection.

Dangerous pattern:

```python
def find_user_by_name(conn, username):
    # ❌ vulnerable: user can inject SQL
    sql = f"SELECT id, username FROM users WHERE username = '{username}'"
    cur = conn.cursor()
    cur.execute(sql)
    return cur.fetchall()
```

If `username` is:

```
alice' OR 1=1 --
```

the resulting SQL becomes:

```
SELECT id, username FROM users WHERE username = 'alice' OR 1=1 --'
```

This may return **all users** instead of just one.

SQL Injection — Safe Pattern with Parameters

Use **parameterized queries** (you started doing this in Module 7).

```python
import sqlite3

def find_user_by_name(conn, username):
    sql = "SELECT id, username FROM users WHERE username = ?"
    cur = conn.cursor()
    cur.execute(sql, (username,))
    return cur.fetchall()

conn = sqlite3.connect(":memory:")
print("This is a demo of parameterized queries, not full setup.")
```

———————————— [finished] ————————————

```
This is a demo of parameterized queries, not full setup.
```

Key rules:

- **Never** build SQL with `+` / f-strings for untrusted input
- Always use `?` placeholders (`%s` in some drivers) and pass **parameters separately**
- Let the database driver escape dangerous characters safely

This same pattern applies to INSERT, UPDATE, DELETE.

———

Validating Before Hitting the Database

Combine **validation** + **parameters** for defense in depth.

Example: new student for the Student Manager (Module 9 mini project):

```python
def validate_student_payload(data):
    name = data.get("name", "").strip()
    email = data.get("email", "").strip()
    course = data.get("course", "").strip()

    if not (3 <= len(name) <= 60):
        raise ValueError("Name must be 3–60 characters.")
    if "@" not in email or "." not in email:
        raise ValueError("Invalid email format.")
    if not course:
        raise ValueError("Course is required.")

    return name, email, course
```

Then in your API endpoint or CLI:

1. Call `validate_student_payload()`
2. Use a **parameterized INSERT** into SQLite
3. Catch `ValueError` and return a friendly message / status code

Logging — Helpful vs Dangerous

Logging is useful for debugging and incident response, but it’s easy to leak secrets.

Common mistakes:

- Logging raw passwords or tokens
- Logging entire HTTP bodies with credentials
- Logging full database rows with sensitive columns (e.g., grades, hashes)

Guidelines:

- **Never log passwords** (even hashes are rarely needed in logs)
- Don’t log API keys, JWTs, or session cookies
- Redact or summarize sensitive fields

Bad:

```
print("Login request:", email, password)
```

Better:

```
print("Login attempt for:", email)
```

Storing Passwords — Concepts

You should almost **never** store plain-text passwords.

Concepts (high level, without full crypto details):

- Use a **hash function designed for passwords** (bcrypt, Argon2, PBKDF2)
- Store only the **salted hash**, never the raw password
- On login, hash the provided password and compare to stored hash

In this course:

- We don’t implement full auth, but you should know:
  - `hashlib.sha256` is **not ideal** for real passwords (too fast)
  - Libraries like `passlib` or frameworks handle this better

Takeaway: if you ever see code writing password strings directly to a database or CSV, that’s a **red flag**.

Config & Secrets Management (Concept)

Where do API keys, DB URLs, and secrets live?

Better options than hard-coding:

- Environment variables (e.g., `os.environ["API_KEY"]`)
- `.env` files loaded via helper libraries (in real projects)
- Separate config files with **restricted permissions**

Bad:

```
API_KEY = "super-secret-key-123"
```

Better (conceptually):

```
import os
API_KEY = os.environ.get("API_KEY")
```

For this course: avoid committing real secrets to code, and keep demo values clearly fake.

Error Messages & Information Leakage

Error messages help users, but can help attackers too.

Examples:

- Revealing exact SQL queries or stack traces in production
- Returning “User exists, but password is wrong” vs a generic message

Guidelines:

- Show **friendly, minimal** messages to end users
- Log detailed technical info to files (with **no secrets**) for developers
- Avoid exposing framework versions, internal paths, or SQL details in HTTP responses

In your CLI/web mini projects, aim for:

> “Something went wrong, please try again or contact support.”
>
> plus a more detailed log entry for yourself.

CLI Tools, Logging & Advanced Python

Python Course · Module 11

ElnurBDa

Module 11 — Command-Line Tools & Advanced Python

In this module we:

- Build **real command-line interfaces (CLIs)** using `argparse`
- Replace `input()`-only scripts with tools that accept flags and subcommands
- Introduce **logging** instead of `print()` for serious scripts
- Explore a few **advanced Python features** you will see in real projects

Focus: make your scripts feel like **professional tools** and understand patterns you'll meet in open-source and production code.

From Scripts to Tools

So far many examples used:

```
name = input("Enter your name: ")
print("Hello", name)
```

This is great for learning, but less ideal when you want to:

- Automate tasks from other scripts or CI
- Run jobs from cron / Task Scheduler
- Reuse the same script with different options

Command-line tools use **arguments** and **flags** instead:

```
python greet.py --name Alice
```

We'll use argparse from the standard library to build these.

Basics of argparse

Pattern:

1. Create a parser
2. Define arguments
3. Parse `sys.argv` into a friendly object

```python
import argparse

def main():
    parser = argparse.ArgumentParser(
        description="Greet someone from the command line."
    )
    parser.add_argument("--name", "-n", required=True, help="Name to greet")

    args = parser.parse_args()

    print(f"Hello, {args.name}!")

if __name__ == "__main__":
    main()
```

———————————————— [finished with error] ————————————————

```
usage: snippet.py [-h] --name NAME
snippet.py: error: the following arguments are required: --name/-n
```

Usage:

```
python greet.py --name Alice
python greet.py -n Bob
```

Try running with `--help` to see automatically generated help text.

Flags, Options & Positional Arguments

argparse supports:

- **Positional args** — required, order matters
- **Options / flags** — usually start with -- or -

Example:

```python
import argparse

def main():
    parser = argparse.ArgumentParser(description="Convert kilometers to miles.")
    parser.add_argument("kilometers", type=float, help="Distance in km")
    parser.add_argument(
        "--precision",
        "-p",
        type=int,
        default=2,
        help="Number of decimal places (default: 2)",
    )

    args = parser.parse_args()
    miles = args.kilometers * 0.621371
    print(f"{args.kilometers} km = {miles:.{args.precision}f} miles")

if __name__ == "__main__":
    main()
```

———————————————— [finished with error] ————————————————

```
usage: snippet.py [-h] [--precision PRECISION] kilometers
snippet.py: error: the following arguments are required: kilometers
```

Usage:

```
python convert.py 10
python convert.py 10 --precision 4
```

———

Subcommands (Like git commit, git push)

Many tools have **subcommands**:

- `git status`, `git commit`, `git push`
- `docker run`, `docker ps`

We can model this with **sub-parsers**.

```python
import argparse

def cmd_add(args):
    print("Adding task:", args.title)

def cmd_list(args):
    print("Listing tasks; completed =", args.completed)

def build_parser():
    parser = argparse.ArgumentParser(prog="tasks", description="Simple task CLI")
    subparsers = parser.add_subparsers(dest="command", required=True)

    add_p = subparsers.add_parser("add", help="Add a new task")
    add_p.add_argument("title", help="Title of the task")
    add_p.set_defaults(func=cmd_add)

    list_p = subparsers.add_parser("list", help="List tasks")
    list_p.add_argument(
        "--completed",
        action="store_true",
        help="Show only completed tasks",
    )
    list_p.set_defaults(func=cmd_list)

    return parser

def main():
    parser = build_parser()
    args = parser.parse_args()
    args.func(args)  # dispatch to the right handler

if __name__ == "__main__":
    main()
```

———————————————————— [finished with error] ————————————————————

```
usage: tasks [-h] {add,list} ...
tasks: error: the following arguments are required: command
```

Usage:

Connecting CLIs to Files & Databases

You can wire argparse into your existing **file** and **SQLite** code:

- tasks.py add → insert into SQLite (Module 7)
- tasks.py export --format csv → read rows and write CSV (Module 6)
- students.py deactivate --email student@example.com → update database row

Pattern:

- Each subcommand calls into a **function** in another module (db.py, services.py, etc.).
- CLI layer **parses arguments**, then passes them to core logic.

This keeps your business logic **testable** and your CLI thin.

Mini Task – Log Tracker CLI

Extend your **log tracker** project (Module 6) into a CLI:

- Create `log_cli.py` with `argparse`:
  - Subcommands: `add`, `list`, `clear`
  - Flags: `--user`, `--action`, maybe `--level` (INFO/WARN/ERROR)
- Internally, call functions from a separate module (e.g., `log_core.py`) that:
  - open the log file
  - append or read lines
  - handle errors gracefully

Goal: run commands like:

```
python log_cli.py add --user alice --action "login"
python log_cli.py list
python log_cli.py clear
```

Why Logging Beats print()

`print()` is fine while learning, but real programs use **logging**:

- Different **levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- Can write to **files**, not just the console
- Easier to turn on/off detailed output without changing code

Python's `logging` module is built-in and flexible.

Logging Basics

Simple pattern:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
)

logging.debug("This is a debug message")
logging.info("Starting application")
logging.warning("Low disk space")
logging.error("Something went wrong")
```

```
──────────────────── [finished] ────────────────────
```

```
2025-11-28 18:34:25,012 [INFO] Starting application
2025-11-28 18:34:25,012 [WARNING] Low disk space
2025-11-28 18:34:25,012 [ERROR] Something went wrong
```

Notes:

- `basicConfig` sets a **global** configuration for the root logger
- By default, output goes to **stderr**
- Change `level=` to control how verbose logs are

Try changing `level=logging.DEBUG` to see all messages.

Log Levels in Practice

Common use:

- DEBUG – very detailed, for developers only
- INFO – high-level events (startup, shutdown, key actions)
- WARNING – something unexpected, but program continues
- ERROR – serious problem, part of the feature failed
- CRITICAL – very bad, program may exit

Example:

```python
import logging

logging.basicConfig(level=logging.DEBUG)

def divide(a, b):
    logging.debug("divide called with a=%s, b=%s", a, b)
    if b == 0:
        logging.error("Attempted division by zero!")
        return None
    return a / b

result = divide(10, 0)
logging.info("Result is %s", result)
```

———————————— [finished] ————————————

```
DEBUG:root:divide called with a=10, b=0
ERROR:root:Attempted division by zero!
INFO:root:Result is None
```

Note the %s style placeholders – logging handles formatting lazily.

Logging to a File

Instead of writing only to the console, log to a **file**:

```python
import logging

logging.basicConfig(
    filename="app.log",
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(name)s: %(message)s",
)

logger = logging.getLogger("demo")

logger.info("Application started")
logger.warning("This is a warning")
logger.error("This is an error")
```

──────────────────────────── [finished] ────────────────────────────

Open `app.log` in your editor to inspect messages.

Tips:

- Use `getLogger(__name__)` in real modules for better names
- Combine with security best practices (Module 10): **do not** log secrets

---

Mini Task – Add Logging to an Existing Project

Pick one earlier project:

- File log tracker (Module 6)
- SQLite score tracker (Module 7)
- Student Manager / To-Do backend (Module 9)

Add logging:

1. Configure `logging` in the main entry point.
2. Replace key `print()` calls with appropriate log levels.
3. Ensure that **no sensitive info** (passwords, tokens) is logged.
4. Trigger some errors to see how they appear in logs.

Advanced Python — Comprehensions Recap

You've seen list comprehensions. Quick recap:

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
evens = [n for n in nums if n % 2 == 0]

print("squares:", squares)
print("evens:", evens)
```

——————————— [finished] ———————————

```
squares: [1, 4, 9, 16, 25]
evens: [2, 4]
```

Two more related tools:

- **Dict comprehensions**
- **Generator expressions**

We'll see both next.

Dict Comprehensions

Build dictionaries in one expression:

```
users = ["alice", "bob", "charlie"]

lengths = {name: len(name) for name in users}
print(lengths)
```

```
──────────────────── [finished] ────────────────────
```

```
{'alice': 5, 'bob': 3, 'charlie': 7}
```

You can also transform existing dicts:

```
scores = {"alice": 90, "bob": 75, "charlie": 82}

passed = {name: s for name, s in scores.items() if s >= 80}
print(passed)
```

```
──────────────────────── [finished] ────────────────────────
```

```
{'alice': 90, 'charlie': 82}
```

Use case: quick mappings, filtered views, basic data transformations.

Generator Expressions

List comprehensions materialize a list in memory. **Generator expressions** produce values **on demand**:

```
nums = range(1, 1_000_001)

total = sum(n * n for n in nums)
print("Sum of squares from 1 to 1,000,000:", total)
```

———————————— [finished] ————————————

```
Sum of squares from 1 to 1,000,000:
333333833333500000
```

Key differences:

- `[...]` → list comprehension (immediate list)
- `(...)` → generator expression (lazy)

Use generators when:

- The result is large
- You only need to **iterate once** or feed another function like `sum`, `max`

---

*args and **kwargs

Sometimes you don’t know in advance how many arguments a function needs.

- *args – extra **positional** arguments (tuple)
- **kwargs – extra **keyword** arguments (dict)

```
def debug_print(*args, **kwargs):
    print("ARGS:", args)
    print("KWARGS:", kwargs)

debug_print(1, 2, 3, name="alice", active=True)
```

———————————— [finished] ————————————

```
ARGS: (1, 2, 3)
KWARGS: {'name': 'alice', 'active': True}
```

Typical uses:

- Wrapper functions that pass arguments through
- Flexible APIs where some options are optional/rare

Default Parameters & Keyword-Only Arguments

You've seen simple defaults:

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

greet("Alice")
greet("Bob", greeting="Hi")
```

———————— [finished] ————————

```
Hello, Alice!
Hi, Bob!
```

We can also force arguments to be **keyword-only** (for clarity):

```
def create_user(username, *, is_admin=False):
    print("Creating user:", username, "is_admin:", is_admin)

create_user("alice")
create_user("bob", is_admin=True)
# create_user("charlie", True)  # TypeError: must use keyword
```

———————— [finished] ————————

```
Creating user: alice is_admin: False
Creating user: bob is_admin: True
```

Use keyword-only args when positional use would be confusing.

Simple Decorators (Concept)

Decorators are a powerful pattern for **wrapping** functions with extra behavior.

Example: log when a function is called.

```python
import time

def log_calls(func):
    def wrapper(*args, **kwargs):
        print(f"[log_calls] calling {func.__name__} with args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"[log_calls] {func.__name__} returned {result}")
        return result
    return wrapper

@log_calls
def add(a, b):
    return a + b

add(2, 3)
```

———————————————————— [finished] ————————————————————

```
[log_calls] calling add with args=(2, 3), kwargs={}
[log_calls] add returned 5
```

Pattern:

1. Decorator takes a function (`func`)
2. Returns a new function (`wrapper`) that adds behavior
3. `@decorator_name` syntax applies it to `add`

———

Timing Functions with a Decorator

Another common use: measure how long a function takes.

```python
import time

def timed(func):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        duration = (time.perf_counter() - start) * 1000
        print(f"{func.__name__} took {duration:.2f} ms")
        return result
    return wrapper

@timed
def slow_operation():
    time.sleep(0.2)

slow_operation()
```

———————————— [finished] ————————————

```
slow_operation took 200.08 ms
```

Real projects use decorators for:

- Logging
- Caching
- Authorization checks
- Retrying failed operations

Mini Project – CLI Admin Tool

Tie everything together by building a **CLI admin tool** for one of your apps:

- Example: Student Manager (`students` table in SQLite)
- Or: To-Do app tasks (Module 9)

Requirements:

1. Use `argparse` with subcommands like `list`, `create`, `deactivate`, `stats`.
2. Connect subcommands to database functions (reuse Module 7 patterns).
3. Use `logging` to record actions (who, what, when) to a file.
4. Use comprehensions / generator expressions for simple summaries (e.g., count active students).
5. Add at least one decorator (`@timed` or `@log_calls`) around a database operation.

You now have the pieces to turn small scripts into **serious tools** that are debuggable, observable, and maintainable.
