# Logging and Testing in Python

PALLAVI VAIDYA

# Agenda

- Basic Logging
- Why to use?
- Where to use?
- Programs

# Logging

**Logging** in Python (and in general programming) means **recording events while your program runs.**

# Why to use Logging?

- Debugging
- Monitoring
- Error Tracking
- Control

# Levels of Logging

DEBUG → Detailed info (good for developers).

INFO → Confirmation things are working normally.

WARNING → Something unexpected, but program still runs.

ERROR → A problem that prevented part of the program from working.

CRITICAL → A very serious error, program may not continue.

# Pitfall 1: Using print()

print() cannot filter levels (INFO, ERROR, etc.)

Cannot easily redirect to files

Not suitable for production systems

# Pitfall 2: Wrong Log Level

If level is set too high (e.g., WARNING), INFO/DEBUG logs are hidden

Developers may think logging is broken

Solution: Use DEBUG in development, INFO/ERROR in production

# Pitfall 3: Multiple basicConfig() Calls

basicConfig() only works the first time

Subsequent calls are ignored

Solution: use custom logger + handlers instead

# Pitfall 4: Forgetting Flush/Close

Logs are written lazily (buffered)

If program crashes, logs may be lost

Solution: call logging.shutdown() at program exit

# Pitfall 5: Logging Sensitive Data

Avoid logging passwords, tokens, or personal info

Risk of leaking sensitive data to files

Solution: mask or filter sensitive values

# Pitfall 6: Too Much Logging

DEBUG everywhere can flood logs

Huge files, hard to find useful info

Solution: use proper levels and filter logs

# Pitfall 7: No Log Rotation

app.log can grow to gigabytes

Can fill disk and crash system

Solution: use RotatingFileHandler or TimedRotatingFileHandler

# Pitfall 8: Logging Inside Loops

Logging inside tight loops slows performance

Floods logs with repetitive messages

Solution: throttle or sample log messages

# Pitfall 9: Multithreading Issues

Logs from multiple threads may interleave

Makes debugging difficult

Solution: structured logging (JSON, key=value)

# Best Practices

Use logging instead of print()

Choose the right log level

Avoid sensitive data in logs

Rotate logs to prevent large files

Balance detail vs. performance

Use structured logging in distributed systems

# RESTful API

**REST** stands for **Representational State Transfer.**

A **RESTful API** is an **API (Application Programming Interface)** that follows REST principles to allow communication between client and server over **HTTP**.

# RESTful API

RESTful APIs are **stateless** → the server does not store client state between requests.

They use **resources** → each piece of data (like a user, product, or order) is a resource identified by a **URL**.

Communication is usually **JSON** or **XML**.

RESTful APIs use **HTTP methods** to perform actions on resources.

# Core HTTP Methods in REST

| HTTP Method | Purpose | REST Operation | Example |
|---|---|---|---|
| **GET** | Retrieve data | Read | `GET /users` → get all users |
| **POST** | Create new data | Create | `POST /users` → create a new user |
| **PUT** | Update existing data (full update) | Update | `PUT /users/123` → update user 123 |
| **PATCH** | Update existing data (partial update) | Update | `PATCH /users/123` → update only certain fields of user 123 |
| **DELETE** | Remove data | Delete | `DELETE /users/123` → delete user 123 |

# JSON

**JSON** = JavaScript Object Notation

Lightweight data format for **storing and exchanging data**

Easy for humans to **read/write** and for machines to **parse/generate**

**Structure:**

**Objects** → key-value pairs, wrapped in { }

**Arrays** → ordered lists, wrapped in [ ]

# Flask

**Flask** is a **micro web framework** in Python.

It allows you to build **web applications and RESTful APIs** quickly.

“Micro” means it doesn’t include extra tools by default (like database ORM, authentication). You can add them as needed.

Flask is simple and beginner-friendly.

# Flask

Lightweight and easy to learn.

Flexible, you can add extensions for database, authentication, etc.

Routes are defined using `@app.route`.

JSON responses are returned using `jsonify`.

# jsonify

It converts Python data (like lists and dicts) into **JSON format** for APIs.

# app

`app` is an **object** created from the framework class (`Flask()` in Flask, `FastAPI()` in FastAPI).

It acts as your **server** that:

Listens for **HTTP requests** (GET, POST, PUT, DELETE, etc.)

Routes requests to the appropriate **function** (endpoint/route)

Sends **HTTP responses** back to the client

# What is FastAPI?

**FastAPI** is a modern, fast (high-performance) **Python web framework** for building **APIs**, especially **RESTful APIs**.

It is designed to be:

**Fast**

- Very high performance, on par with NodeJS and Go.
- Uses **ASGI** (Asynchronous Server Gateway Interface) for async programming.

**Easy to Use / Developer-Friendly**

- Simple syntax, similar to Flask.
- Automatic **data validation** and **type checking** using **Pydantic**.

# What is FastAPI?

**Automatic Documentation**

- Generates **interactive API docs** automatically with **Swagger UI**:
  Visit http://127.0.0.1:8000/docs

**Supports Standard Python Type Hints**

- You define the type of your parameters (int, str, List[str]) → FastAPI validates inputs automatically.

**Asynchronous Support**

- Can handle async requests using async def for better performance with many requests.

# Why Use FastAPI?

Perfect for building **REST APIs**, microservices, or backend services.

Handles **JSON input/output** easily.

Built-in validation prevents many common bugs.

Supports **async programming** for high-performance APIs.

FastAPI **automatically converts list to JSON**, validates data, and supports async.

Flask requires `jsonify` and manual validation.

# uvicorn

Uvicorn is a fast Asynchronous Server Gateway Interface (ASGI) web server implementation for Python. In the context of FastAPI, Uvicorn acts as the server that runs your FastAPI application.

It runs your FastAPI app so it can respond to HTTP requests.

# uvicorn

**Client** → sends HTTP request

**ASGI server** ( Uvicorn) → receives request

**Web application** (like FastAPI) → processes request, returns response

ASGI server → sends response back to client

So ASGI is like a **middleman** that understands both **async code** and **HTTP protocol**.

# Unit Testing in Python

Unit testing in Python is testing **smallest parts of your code (functions or methods)** to ensure they behave as expected.

**Key Module:** `unittest` (built-in Python library)

# Basic Steps for Unit Testing in Python

**Import `unittest`**

**Define a test class** inheriting from `unittest.TestCase`

**Write test methods** (methods starting with `test_`)

**Use assertions** to check expected vs actual results

**Run tests** using `unittest.main()`

# Basic Steps for Unit Testing in Python

| Assertion | Usage |
|---|---|
| `assertEqual(a, b)` | Check if a == b |
| `assertNotEqual(a, b)` | Check if a != b |
| `assertTrue(x)` | Check if x is True |
| `assertFalse(x)` | Check if x is False |
| `assertIsNone(x)` | Check if x is None |
| `assertIsNotNone(x)` | Check if x is not None |
| `assertRaises(Exception)` | Check if exception is raised |

# Tips for Python Unit Testing

Test **one function at a time**.

Include **edge cases** (empty list, zero, negative numbers).

Keep **tests independent** — one test should not depend on another.

Name test methods **clearly** (`test_add_positive_numbers`).

Run tests frequently during development.