PYTHON LOGGING MODULE

WHAT IS LOGGING IN PYTHON?

Logging = recording events that happen while a program runs.

- Used to:
  - Debug and understand program flow
  - Monitor behavior in production
  - Diagnose errors after they happen (post-mortem)
  - Audit actions (e.g., who did what and when)
- Python provides a built-in **logging** module → no extra install needed.

2. WHY LOGGING INSTEAD OF PRINT()?

print():

- Only writes to stdout
- Hard to control what gets printed and where
- Hard to turn off in production without editing code

logging:

- **Levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- Can log to **console, files, rotating files, sockets, emails, etc.**
- Central configuration: change behavior without changing all call sites
- Supports **timestamps, module name, line number, etc.**

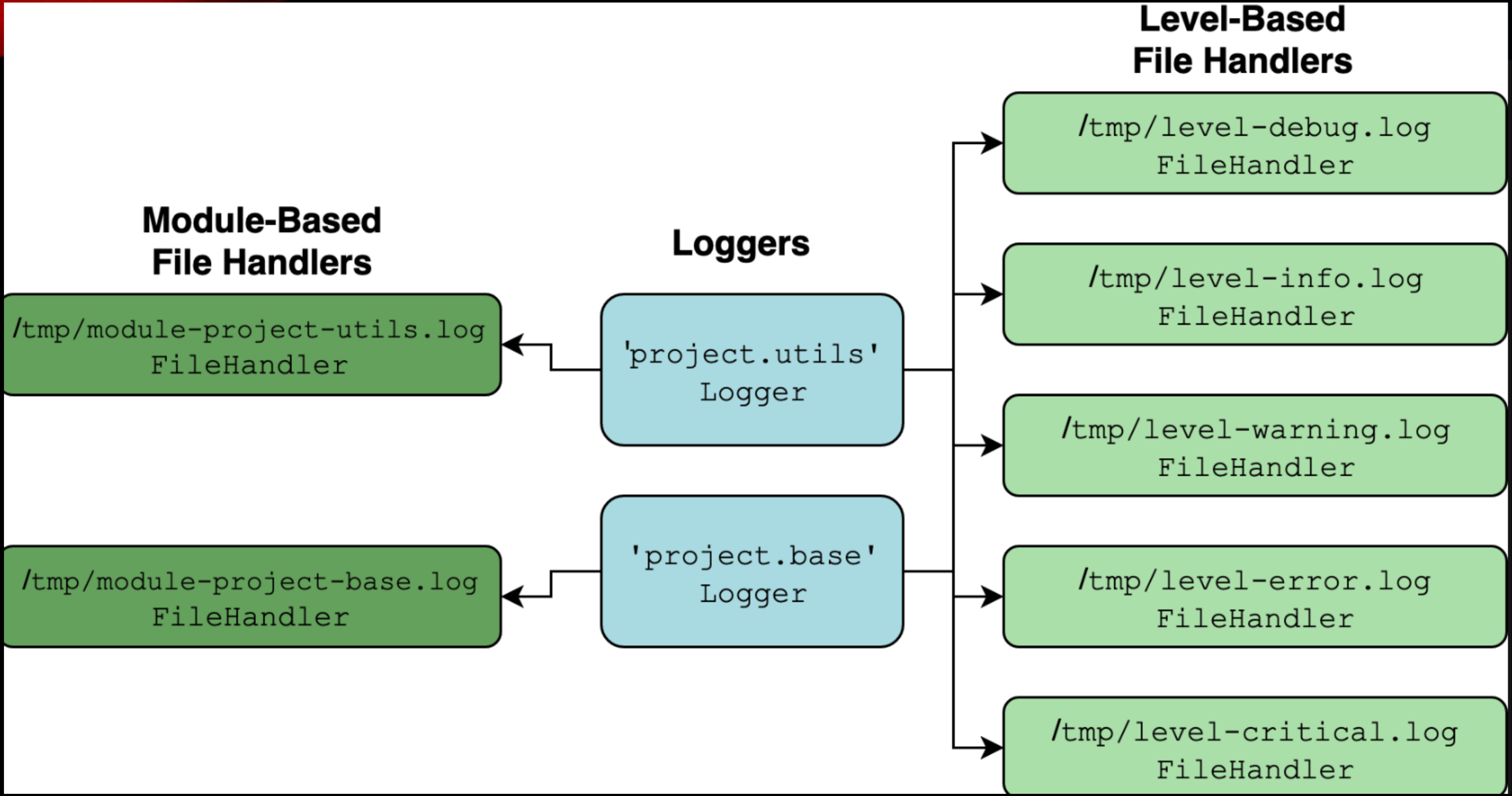
3. CORE CONCEPTS IN THE LOGGING MODULE AND TYPES OF LOGGING

- Think of the logging system as a **pipeline**:
- **Logger**
  - Entry point for your code.
  - You call methods like logger.debug(), logger.info(), etc.
  - Usually obtained via:

```
import logging

logging.debug("Detailed info for developers")       # Level 10
logging.info("General operational confirmation")    # Level 20
logging.warning("Something unexpected happened")    # Level 30 ← DEFAULT
logging.error("A serious problem occurred")         # Level 40
logging.critical("Very severe error – may crash")   # Level 50
```

MULTIPRONGED LOGGING



3. Basic Configuration with basicConfig()

```
import logging

# Configure once at the start of your program
logging.basicConfig(
    level=logging.INFO,                                        # Show INFO
and above
    format="%(asctime)s - %(levelname)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    filename="app.log",                                  # Optional: log to file
    filemode="a"                                               # Append mode
)

logging.info("Application started")
logging.warning("Disk space is low!")
```

4. Using Named Loggers (Best Practice)

```
import logging

# Create a logger with module name
logger = logging.getLogger(__name__)

def process_data():
    logger.info("Starting data processing")
    logger.debug("Raw data: %s", data)
    try:
        # ... processing logic ...
        logger.info("Data processed successfully")
    except Exception as e:
        logger.error("Failed to process data: %s", str(e))
```

HANDLER

Decides where the log goes (destination).

- Examples:
- StreamHandler (console)
- FileHandler (plain file)
- RotatingFileHandler (log rotation by size)
- TimedRotatingFileHandler (rotation by time)
- SMTPHandler (send logs via email)
- SocketHandler, HTTPHandler, etc.
- Each handler:
- Has its own level
- Has Formatter and Filter(s)

5. LOGGING TO DIFFERENT HANDLERS

```
import logging
import sys

logger = logging.getLogger("pylog")
logger.setLevel(logging.DEBUG)
h1 = logging.FileHandler(filename="/tmp/records.log")
h1.setLevel(logging.INFO)
h2 = logging.StreamHandler(sys.stderr)
h2.setLevel(logging.ERROR)
logger.addHandler(h1)
logger.addHandler(h2)
logger.info("testing %d.. %d.. %d..", 1, 2, 3)
```

REAL-WORLD EXAMPLE: ML TRAINING PIPELINE

```
import logging

# Configure
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(levelname)s - %(funcName)s - %(message)s"
)
logger = logging.getLogger(__name__)

def load_data():
    logger.info("Loading dataset from S3")
    # ... load ...
    logger.debug("Dataset shape: (1000, 20)")
    return data

def train_model(data):
    logger.info("Starting model training")
    try:
        # ... train ...
        logger.info("Model trained successfully. Accuracy: 0.92")
    except Exception as e:
        logger.critical("Training failed: %s", str(e))
        raise

if __name__ == "__main__":
    data = load_data()
    train_model(data)
```