**Step-by-Step Guide to Setting Up Logging for a Machine Learning Python Application**

Logging is essential for tracking the execution of your ML application, debugging issues, and monitoring model performance. Below is a structured approach to setting up logging in a Python-based ML project.

**Step 1: Import Required Libraries**

First, you need to import the logging module.

python

CopyEdit

import logging

import os

**Step 2: Create a Log Directory (Optional)**

To store logs in files, you may want to create a dedicated directory.

python

CopyEdit

log\_dir = "logs"

os.makedirs(log\_dir, exist\_ok=True)

log\_file = os.path.join(log\_dir, "ml\_app.log")

**Step 3: Configure Logging**

Set up logging with the required format, log level, and file handler.

python

CopyEdit

def setup\_logger():

"""Configures logging settings."""

logging.basicConfig(

level=logging.INFO, # Set logging level: DEBUG, INFO, WARNING, ERROR, CRITICAL

format="[{asctime}] {levelname} - {message}",

datefmt="%Y-%m-%d %H:%M:%S",

style="{",

handlers=[

logging.FileHandler(log\_file), # Log to a file

logging.StreamHandler() # Log to console

]

)

setup\_logger()

logger = logging.getLogger(\_\_name\_\_) # Create logger instance

**Step 4: Add Logging Statements in Your ML Code**

Now, use logging to track different stages of your ML pipeline.

**Example: Logging in a Machine Learning Pipeline**

python

CopyEdit

import pandas as pd

import numpy as np

# Log start of data loading

logger.info("Loading dataset...")

data = pd.DataFrame(np.random.rand(100, 2), columns=["feature1", "feature2"])

logger.info(f"Dataset loaded with shape: {data.shape}")

# Log start of preprocessing

logger.info("Starting data preprocessing...")

data.fillna(data.mean(), inplace=True)

logger.info("Preprocessing complete.")

# Log model training

logger.info("Training model...")

# Example: Placeholder for model training

logger.info("Model training completed successfully.")

# Log model evaluation

logger.info("Evaluating model...")

# Example: Placeholder for model evaluation

logger.info("Model evaluation completed.")

**Step 5: Run and Check Logs**

* Logs will appear in both the **console** and the **logs/ml\_app.log** file.
* Example log output:

csharp

CopyEdit

[2025-03-07 12:30:01] INFO - Loading dataset...

[2025-03-07 12:30:02] INFO - Dataset loaded with shape: (100, 2)

[2025-03-07 12:30:03] INFO - Starting data preprocessing...

[2025-03-07 12:30:04] INFO - Preprocessing complete.

[2025-03-07 12:30:05] INFO - Training model...

[2025-03-07 12:30:06] INFO - Model training completed successfully.

**Step 6: Customize Log Levels for Better Debugging**

* **logger.debug("Debugging information")** → For detailed debugging.
* **logger.info("General information")** → For tracking progress.
* **logger.warning("Warning message")** → For potential issues.
* **logger.error("Error message")** → For critical errors.
* **logger.critical("Critical issue")** → For severe problems.

**Step 7: (Optional) Use logging.config for Advanced Logging**

For more complex setups (like logging to multiple files, JSON formatting), use logging.config.

python

CopyEdit

import logging.config

logging.config.fileConfig('logging.conf') # Load from a config file

logger = logging.getLogger("ml\_pipeline")

**Conclusion**

By following these steps, you can efficiently log events in your ML application, making debugging and monitoring much easier.

**Advanced Logging Configuration (logging.conf)**

A logging.conf file allows you to configure logging in a structured way without modifying the Python code directly. Below is a configuration file that:

* Logs messages to both a file and the console
* Uses different log levels
* Rotates log files to prevent excessive file sizes

**1️⃣ Create a logging.conf File**

Save the following content as logging.conf in your project directory.

ini

CopyEdit

[loggers]

keys=root,ml\_pipeline

[handlers]

keys=consoleHandler,fileHandler

[formatters]

keys=simpleFormatter,detailedFormatter

[logger\_root]

level=INFO

handlers=consoleHandler

[logger\_ml\_pipeline]

level=DEBUG

handlers=consoleHandler,fileHandler

qualname=ml\_pipeline

propagate=0

[handler\_consoleHandler]

class=StreamHandler

level=INFO

formatter=simpleFormatter

args=(sys.stdout,)

[handler\_fileHandler]

class=logging.handlers.RotatingFileHandler

level=DEBUG

formatter=detailedFormatter

args=('logs/ml\_pipeline.log', 'a', 10485760, 5)

[formatter\_simpleFormatter]

format=[%(asctime)s] %(levelname)s - %(message)s

datefmt=%Y-%m-%d %H:%M:%S

[formatter\_detailedFormatter]

format=[%(asctime)s] %(levelname)s %(module)s:%(lineno)d | %(message)s

datefmt=%Y-%m-%d %H:%M:%S

**2️⃣ Update Your Python Code to Use logging.conf**

Modify your ML application to load this configuration:

python

CopyEdit

import logging

import logging.config

import os

# Ensure logs directory exists

os.makedirs("logs", exist\_ok=True)

# Load logging configuration from file

logging.config.fileConfig("logging.conf")

# Create a logger instance

logger = logging.getLogger("ml\_pipeline")

# Example logging usage

logger.debug("This is a debug message (useful for debugging)")

logger.info("Pipeline started successfully")

logger.warning("Potential issue detected")

logger.error("An error occurred while processing data")

logger.critical("Critical issue in the pipeline")

**3️⃣ Explanation of logging.conf**

| **Section** | **Purpose** |
| --- | --- |
| [loggers] | Defines available loggers (root, ml\_pipeline) |
| [handlers] | Specifies handlers (console and file) |
| [formatters] | Defines different log formats |
| [logger\_ml\_pipeline] | Logs messages from ML pipeline with DEBUG level and above |
| [handler\_consoleHandler] | Prints logs to the console (sys.stdout) |
| [handler\_fileHandler] | Writes logs to logs/ml\_pipeline.log (rotates after 10MB, keeps 5 backups) |
| [formatter\_simpleFormatter] | Simple log format for console |
| [formatter\_detailedFormatter] | Detailed log format for file (includes module name and line number) |

**4️⃣ Features of This Setup**

✅ Logs both to **console** and **file**  
✅ Uses **rotating logs** (max 10MB per file, keeps last 5 logs)  
✅ Provides **detailed formatting** (with timestamp, log level, module, and line number)  
✅ Different **log levels** for filtering messages

**5️⃣ Sample Log Output**

**Console Output (simpleFormatter)**

csharp

CopyEdit

[2025-03-07 14:00:12] INFO - Pipeline started successfully

[2025-03-07 14:00:14] WARNING - Potential issue detected

**Log File (detailedFormatter in logs/ml\_pipeline.log)**

less

CopyEdit

[2025-03-07 14:00:12] INFO main:23 | Pipeline started successfully

[2025-03-07 14:00:14] WARNING data\_loader:56 | Potential issue detected

**Final Thoughts**

By using logging.conf, you separate logging configuration from code, making it **cleaner, reusable, and more customizable**.

**Difference Between print() and logging in Python**

| **Feature** | **print()** | **logging** |
| --- | --- | --- |
| **Purpose** | Debugging, displaying output | Structured logging for debugging, monitoring, and error tracking |
| **Output Control** | Always prints to console | Can log to console, files, databases, or external monitoring systems |
| **Severity Levels** | No built-in levels | Has levels: DEBUG, INFO, WARNING, ERROR, CRITICAL |
| **Configurability** | Not configurable | Highly configurable (handlers, formatters, filters) |
| **Persistence** | No automatic storage | Logs can be stored, rotated, and analyzed later |
| **Performance Impact** | Can slow down code if used excessively | More efficient in large applications with proper configuration |
| **Use Case** | Simple debugging, small scripts | Production-ready applications, debugging, error tracking |

**When to Use What?**

* ✅ **Use print()** when debugging small scripts or temporary testing.
* ✅ **Use logging** for **serious applications**, monitoring, debugging, and tracking issues in production.

**Example**

python

CopyEdit

import logging

# Configure logging

logging.basicConfig(level=logging.INFO)

print("Script started") # Prints to console

logging.info("Script started") # Logs with INFO level

try:

x = 10 / 0

except ZeroDivisionError:

print("Error occurred!") # Just prints

logging.error("Division by zero occurred", exc\_info=True) # Logs error with traceback

**Output**

**print() Output:**

javascript

CopyEdit

Script started

Error occurred!

**logging Output:**

vbnet

CopyEdit

INFO:root:Script started

ERROR:root:Division by zero occurred

Traceback (most recent call last):

File "script.py", line 8, in <module>

x = 10 / 0

ZeroDivisionError: division by zero

**Conclusion**

* print() is **basic** and should be used for quick checks.
* logging is **powerful, structured, and scalable**—ideal for real-world applications.