Python Logging Module
Complete Reference Guide

For MLOps and Production Systems

A Comprehensive Guide to Logging,
Best Practices, and Real-World Implementation

Sujil S

sujil9480@gmail.com

December 25, 2025

Contents

1 Introduction to Python Logging

1.1 What is Logging?

Logging is the process of recording events, messages, and information during program execution. It provides a way to track what happens when software runs, which is essential for understanding application behavior, debugging issues, and monitoring systems in production.

1.1.1 Why Logging Matters

Logging is critical for:

- **Debugging and Troubleshooting**: Identify and fix bugs efficiently
- **Monitoring**: Track application behavior in production
- **Auditing**: Maintain records for compliance and security
- **Performance Analysis**: Identify bottlenecks and optimization opportunities
- **Understanding Flow**: Trace execution path through complex systems

1.2 Why Use Logging in MLOps?

In Machine Learning Operations (MLOps), logging becomes even more critical:

1. **Model Training Monitoring**:
    - Track training epochs and iterations
    - Record loss values and metrics
    - Monitor convergence and performance
    - Log hyperparameters for reproducibility
2. **Data Pipeline Tracking**:
    - Monitor data ingestion processes
    - Log preprocessing steps
    - Track data quality issues
    - Record transformation operations
3. **Model Serving**:
    - Log prediction requests and responses
    - Track API performance
    - Monitor model confidence scores
    - Record error rates
4. **Error Tracking**:
    - Identify failures quickly
    - Debug production issues
    - Track error patterns
    - Generate alerts for critical errors

5. **Performance Monitoring**:
    - Track inference times
    - Monitor resource usage (CPU, GPU, memory)
    - Identify performance degradation
    - Optimize system bottlenecks
6. **Reproducibility**:
    - Maintain audit trails for experiments
    - Document model versions and configurations
    - Track data versions used
    - Enable experiment reproduction

1.3 Logging vs Print Statements

Many beginners use `print()` statements for debugging. However, in production code, the logging module is vastly superior:

print() Statements	**logging Module**
Always outputs to console	Flexible output destinations (file, console, remote)
No severity levels	Five severity levels (DEBUG to CRITICAL)
Difficult to filter messages	Easy filtering by level
No automatic timestamps	Automatic timestamps available
Cannot be disabled easily	Can be enabled/disabled by configuration
Not suitable for production	Production-ready and battle-tested
No structured format	Consistent, structured format
Limited context information	Rich context (filename, line number, function)
Performance impact on I/O	Optimized performance with lazy evaluation
No rotation capabilities	Automatic log file rotation

Warning

Production Code Rule: Never use `print()` statements in production code. Always use the logging module for better control, flexibility, and maintainability.

2 Core Logging Components

2.1 The Logging Architecture

The Python logging module follows a hierarchical architecture with four main components that work together to provide flexible and powerful logging capabilities.

1. **Loggers**: Entry point for logging messages in your application
2. **Handlers**: Determine where log records are sent (console, file, network)
3. **Formatters**: Define the layout and content of log messages
4. **Filters**: Provide fine-grained control over which log records are processed

2.2 Logger Objects

The **Logger** is the primary interface that applications use to write log messages.

2.2.1 Creating a Logger

```
import logging

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

# Create a logger with custom name
logger = logging.getLogger('my_application')

# Get the root logger
root_logger = logging.getLogger()
```

Important Note

Best Practice: Always use `__name__` when creating loggers. This creates a hierarchical naming structure based on your module organization, making it easier to manage logging across large applications.

2.2.2 Logger Methods

Loggers provide methods for different severity levels:

- `logger.debug(msg)` - Detailed diagnostic information for debugging
- `logger.info(msg)` - General informational messages about normal operation
- `logger.warning(msg)` - Warning messages indicating potential issues
- `logger.error(msg)` - Error messages for serious problems
- `logger.critical(msg)` - Critical error messages for severe failures
- `logger.exception(msg)` - Error with full traceback (use in except blocks)

2.3 Handler Objects

Handlers determine where log messages are sent. Multiple handlers can be attached to a single logger.

2.3.1 Common Handler Types

StreamHandler	Sends logs to console (stdout/stderr) - useful for development
FileHandler	Writes logs to a file - basic file logging
RotatingFileHandler	Rotates log files based on size - prevents files from growing too large
TimedRotatingFileHandler	Rotates log files based on time intervals - daily, weekly, etc.
HTTPHandler	Sends logs to HTTP server - for centralized logging
SMTPHandler	Sends logs via email - for critical alerts
SysLogHandler	Sends logs to Unix syslog daemon
SocketHandler	Sends logs over network socket
QueueHandler	Sends logs to queue for async processing

2.4 Formatter Objects

Formatters specify the layout and content of log messages, allowing you to customize how information is displayed.

2.4.1 Common Format Attributes

The following placeholders can be used in format strings:

- `%(asctime)s` - Human-readable timestamp of log record
- `%(name)s` - Name of the logger
- `%(levelname)s` - Logging level (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- `%(message)s` - The actual log message
- `%(filename)s` - Source filename where log was called
- `%(lineno)d` - Line number in source code
- `%(funcName)s` - Function name where log was called
- `%(process)d` - Process ID
- `%(thread)d` - Thread ID
- `%(pathname)s` - Full pathname of source file
- `%(module)s` - Module name (filename without extension)
- `%(levelno)s` - Numeric logging level

2.5 Filter Objects

Filters provide fine-grained control over which log records are processed. They can be attached to both loggers and handlers.

Filters are useful for:

- Filtering by specific attributes
- Adding contextual information
- Implementing complex filtering logic
- Rate limiting log messages

3 Logging Levels

3.1 The Five Standard Levels

Python's logging module defines five severity levels in ascending order of importance. Each level has a numeric value:

Level Name	Numeric Value	Purpose
DEBUG	10	Detailed diagnostic information
INFO	20	Informational messages
WARNING	30	Warning messages
ERROR	40	Error messages
CRITICAL	50	Critical error messages

3.2 DEBUG Level (10)

Purpose: Most detailed information for diagnosing problems during development.

When to Use:

- Development and debugging phases
- Tracking variable values and state
- Understanding detailed program flow
- Loop iterations and function entry/exit
- Detailed API request/response information

DEBUG Level Examples in MLOps

```
logger.debug(f"Loading dataset from {data_path}")
logger.debug(f"Feature shape: {X.shape}, Target shape: {y.shape}")
logger.debug(f"Model parameters: {model.get_params()}")
logger.debug(f"Batch {batch_num}: input_shape={input_data.shape}")
logger.debug(f"GPU memory allocated: {torch.cuda.memory_allocated()}")
```

3.3 INFO Level (20)

Purpose: Confirmation that things are working as expected.

When to Use:

- Application startup and shutdown
- Successful completion of operations
- Milestone achievements
- Configuration information
- Normal operational messages

INFO Level Examples in MLOps

```python
logger.info("Model training started")
logger.info(f"Epoch {epoch}/{total_epochs}: loss={loss:.4f}, accuracy={acc:.4f}")
logger.info("Model saved successfully to disk")
logger.info(f"API server started on port {port}")
logger.info(f"Loaded {len(dataset)} samples from database")
logger.info("Data preprocessing pipeline completed")
```

3.4 WARNING Level (30)

Purpose: Indication of unexpected events or potential problems that don’t prevent the program from working.

When to Use:

- Using deprecated features
- Resource constraints or limitations
- Recoverable errors
- Configuration issues that have fallbacks
- Performance concerns

WARNING Level Examples in MLOps

```python
logger.warning("Missing values detected in dataset")
logger.warning(f"GPU memory usage at {usage}%, close to limit")
logger.warning("Model confidence below threshold: 0.6")
logger.warning("Using default hyperparameters")
logger.warning("Training data smaller than recommended minimum")
logger.warning("Deprecated model architecture detected")
```

3.5 ERROR Level (40)

Purpose: Serious problems that prevented a specific function or operation from executing.

When to Use:

- Failed operations
- Exception handling (non-critical)
- Resource unavailability
- Data validation failures
- API request failures

ERROR Level Examples in MLOps

```python
logger.error(f"Failed to load model from {model_path}")
logger.error("Database connection failed")
logger.error(f"Invalid input shape: expected {exp_shape}, got {actual_shape}")
logger.error("Prediction API returned error 500")
logger.error("Data quality check failed: too many outliers")
logger.error(f"Feature extraction failed for {feature_name}")
```

3.6 CRITICAL Level (50)

Purpose: Very serious errors indicating that the program may not be able to continue running.

When to Use:

- System crashes or imminent crashes
- Data corruption
- Security breaches
- Unrecoverable errors
- Service unavailability

CRITICAL Level Examples in MLOps

```python
logger.critical("Out of memory - cannot continue training")
logger.critical("Model serving endpoint unreachable")
logger.critical("Critical security vulnerability detected")
logger.critical("Data pipeline completely failed")
logger.critical("Unable to connect to required external service")
logger.critical("System configuration corrupted")
```

3.7 Setting Logging Levels

You can set logging levels at both the logger and handler level:

```python
import logging

logger = logging.getLogger(__name__)

# Set logger level (processes messages at this level and above)
logger.setLevel(logging.DEBUG)

# Set handler level (outputs messages at this level and above)
handler = logging.StreamHandler()
handler.setLevel(logging.WARNING)

logger.addHandler(handler)

# These will be processed but not output (handler level is WARNING)
logger.debug("Debug message")   # Not output
logger.info("Info message")     # Not output
```

```python

# These will be both processed and output
logger.warning("Warning message")  # Output
logger.error("Error message")      # Output
```

Level Hierarchy

How Levels Work:

- Logger set to INFO will process INFO, WARNING, ERROR, CRITICAL (but not DEBUG)
- Handler set to ERROR will only output ERROR and CRITICAL messages
- Messages below the set level are completely ignored
- Both logger AND handler must allow a level for it to be output

4 Handlers in Detail

4.1 StreamHandler (Console Handler)

The **StreamHandler** sends log output to streams like `sys.stdout` or `sys.stderr`. This is the most common handler for development.

4.1.1 Basic Usage

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create console handler
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)

# Create formatter
formatter = logging.Formatter(
    ’%(asctime)s - %(name)s - %(levelname)s - %(message)s’
)
console_handler.setFormatter(formatter)

# Add handler to logger
logger.addHandler(console_handler)

# Use the logger
logger.info("This appears in console")
logger.debug("This won’t appear (below handler level)")
```

4.1.2 Directing to stderr

```
import sys
import logging

# Direct to stderr instead of stdout
error_handler = logging.StreamHandler(sys.stderr)
error_handler.setLevel(logging.ERROR)

logger.addHandler(error_handler)
```

4.2 FileHandler

The **FileHandler** writes log messages to a specified file on disk.

4.2.1 Basic Usage

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Create file handler
file_handler = logging.FileHandler(’application.log’)
```

```
file_handler.setLevel(logging.ERROR)

formatter = logging.Formatter(
    '%(asctime)s - %(levelname)s - %(message)s'
)
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)

logger.error("This is written to application.log")
logger.info("This is not written (below handler level)")
```

4.2.2 File Modes

- 'w' - Write mode: Overwrites existing file
- 'a' - Append mode: Appends to existing file (default)

```
# Overwrite mode - starts fresh each time
file_handler = logging.FileHandler('app.log', mode='w')

# Append mode - continues adding to existing file
file_handler = logging.FileHandler('app.log', mode='a')
```

4.3 RotatingFileHandler

The **RotatingFileHandler** automatically rotates log files when they reach a certain size. This is essential for production systems to prevent log files from consuming too much disk space.

4.3.1 Usage Example

```
from logging.handlers import RotatingFileHandler
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Rotate after 10MB, keep 5 backup files
rotating_handler = RotatingFileHandler(
    'app.log',
    maxBytes=10*1024*1024,  # 10 MB
    backupCount=5
)

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
rotating_handler.setFormatter(formatter)
logger.addHandler(rotating_handler)

logger.info("Logging with rotation enabled")
```

How File Rotation Works:

1. When app.log reaches 10MB, it's renamed to app.log.1

2. A new `app.log` file is created for current logging
3. On next rotation: `app.log.1` → `app.log.2`, new log → `app.log.1`
4. This continues until `backupCount` is reached
5. Oldest file is deleted when new rotation occurs beyond `backupCount`

> **Important Note**
>
> **Production Tip**: For production systems handling significant traffic, use RotatingFileHandler with appropriate size limits (10-50 MB) and backup counts (5-10 files) to balance between log retention and disk space.

4.4 TimedRotatingFileHandler

The **TimedRotatingFileHandler** rotates log files at specified time intervals rather than by file size.

4.4.1 Usage Example

```python
from logging.handlers import TimedRotatingFileHandler
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

# Rotate daily at midnight, keep 30 days of logs
timed_handler = TimedRotatingFileHandler(
    'app.log',
    when='midnight',
    interval=1,
    backupCount=30
)

formatter = logging.Formatter(
    '%(asctime)s - %(levelname)s - %(message)s'
)
timed_handler.setFormatter(formatter)
logger.addHandler(timed_handler)

logger.info("Logging with time-based rotation")
```

4.4.2 Common ’when’ Values

Value	Type	Description
’S’	Seconds	Rotate every N seconds
’M’	Minutes	Rotate every N minutes
’H’	Hours	Rotate every N hours
’D’	Days	Rotate every N days
’midnight’	Special	Roll over at midnight
’W0’-’W6’	Weekday	Rotate on specific weekday (0=Monday)

Time-Based Rotation Examples

```python
# Rotate every hour
handler = TimedRotatingFileHandler('app.log', when='H', interval=1)

# Rotate every 6 hours
handler = TimedRotatingFileHandler('app.log', when='H', interval=6)

# Rotate daily at midnight
handler = TimedRotatingFileHandler('app.log', when='midnight')

# Rotate every Monday
handler = TimedRotatingFileHandler('app.log', when='W0')
```

4.5 Multiple Handlers

A single logger can have multiple handlers, each with different levels, formatters, and destinations. This is extremely useful for production systems.

Complete Multi-Handler Setup

```python
import logging
from logging.handlers import RotatingFileHandler

# Create logger
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)  # Logger processes all levels

# Console handler - INFO and above to console
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.INFO)
console_formatter = logging.Formatter(
    '%(levelname)s - %(message)s'
)
console_handler.setFormatter(console_formatter)

# File handler - ERROR and above to file
file_handler = RotatingFileHandler(
    'errors.log',
    maxBytes=5*1024*1024,  # 5MB
    backupCount=3
)
file_handler.setLevel(logging.ERROR)
file_formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - '
    '[%(filename)s:%(lineno)d] - %(message)s'
)
file_handler.setFormatter(file_formatter)

# Debug file handler - all messages
debug_handler = RotatingFileHandler(
    'debug.log',
    maxBytes=10*1024*1024,  # 10MB
    backupCount=5
)
debug_handler.setLevel(logging.DEBUG)
debug_handler.setFormatter(file_formatter)

# Add all handlers to logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)
logger.addHandler(debug_handler)

# Test different levels
logger.debug("Debug message")      # Only in debug.log
logger.info("Info message")        # Console + debug.log
logger.error("Error message")      # Console + errors.log + debug.log
```

This setup provides:

- Quick feedback in console (INFO and above)
- Separate error log for critical issues
- Complete debug log for thorough investigation

5 Formatters and Message Formatting

5.1 Creating Formatters

Formatters control the final output format of log messages. They can include various pieces of information about the log record.

5.1.1 Basic Formatter

```python
import logging

# Simple formatter with common fields
formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

# Apply to handler
handler = logging.StreamHandler()
handler.setFormatter(formatter)
```

5.1.2 Custom Date Format

```python
formatter = logging.Formatter(
    fmt='%(asctime)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'  # Custom date format
)

# Example output: 2024-12-17 14:30:45 - INFO - Message
```

5.2 Common Format Patterns

5.2.1 Simple Format (Development)

Best for quick development and debugging:

```python
formatter = logging.Formatter('%(levelname)s - %(message)s')
```

Output example:

```
INFO - Model training started
ERROR - Failed to load data
```

5.2.2 Standard Format (General Purpose)

Balanced format with essential information:

```python
formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
```

Output example:

```
2024-12-17 14:30:45,123 - ml_model - INFO - Model training started
```

5.2.3 Detailed Format (Production)

Includes file and line information for debugging:

```python
formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - '
    '[%(filename)s:%(lineno)d] - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)
```

Output example:

```
2024-12-17 14:30:45 - ml_model - INFO - [train.py:42] - Model training started
```

5.2.4 MLOps Format (Comprehensive)

Includes process/thread info for distributed systems:

```python
formatter = logging.Formatter(

    '%(asctime)s - [PID:%(process)d TID:%(thread)d] - '
    '%(name)s - %(funcName)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)
```

Output example:

```
2024-12-17 14:30:45 - [PID:12345 TID:67890] - ml_model -
train_model - INFO - Epoch 1 complete
```

5.3 Complete Format Attributes Reference

Attribute	Description
`%(name)s`	Logger name (typically module name)
`%(levelname)s`	Text logging level (DEBUG, INFO, etc.)
`%(levelno)s`	Numeric logging level (10, 20, etc.)
`%(pathname)s`	Full pathname of source file
`%(filename)s`	Filename portion of pathname
`%(module)s`	Module name (filename without .py)
`%(lineno)d`	Line number where log was called
`%(funcName)s`	Function name where log was called
`%(created)f`	Time when LogRecord was created (seconds since epoch)
`%(asctime)s`	Human-readable time
`%(msecs)d`	Millisecond portion of time
`%(relativeCreated)d`	Time in ms since logging module loaded
`%(thread)d`	Thread ID
`%(threadName)s`	Thread name
`%(process)d`	Process ID
`%(message)s`	The logged message

6 Logging Configuration

6.1 Basic Configuration with basicConfig()

The `basicConfig()` function provides a quick way to configure logging for simple applications.

6.1.1 Simple Setup

```python
import logging

# Basic configuration with default settings
logging.basicConfig(level=logging.INFO)

# Use root logger
logging.info("This is an info message")
logging.error("This is an error message")
```

6.1.2 Detailed Configuration

```python
import logging

logging.basicConfig(
    level=logging.DEBUG,
    format=’%(asctime)s - %(name)s - %(levelname)s - %(message)s’,
    datefmt=’%Y-%m-%d %H:%M:%S’,
    filename=’app.log’,
    filemode=’w’  # ’w’ to overwrite, ’a’ to append
)

logging.info("Application started")
logging.debug("Debug information")
```

6.1.3 Multiple Handlers with basicConfig()

```python
import logging

logging.basicConfig(
    level=logging.DEBUG,
    format=’%(asctime)s - %(levelname)s - %(message)s’,
    handlers=[
        logging.FileHandler(’debug.log’),
        logging.StreamHandler()
    ]
)

logging.info("Logs to both file and console")
```

Warning

Important Limitation: `basicConfig()` only works the first time it’s called. Subsequent calls are ignored unless you use the `force=True` parameter (Python 3.8+).

```python
# Python 3.8+ only
logging.basicConfig(level=logging.DEBUG, force=True)
```

6.2 Manual Configuration (Recommended)

For production applications, manual configuration provides more control and flexibility.

6.2.1 Complete Setup Example

```
import logging
from logging.handlers import RotatingFileHandler

def setup_logger(name, log_file, level=logging.INFO):
    """
    Function to setup logger with console and file handlers.

    Args:
        name: Logger name
        log_file: Path to log file
        level: Logging level

    Returns:
        Configured logger object
    """

    # Create logger
    logger = logging.getLogger(name)
    logger.setLevel(level)

    # Prevent duplicate handlers
    if logger.handlers:
        return logger

    # Create formatters
    console_formatter = logging.Formatter(
        '%(levelname)s - %(message)s'
    )
    file_formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - '
        '[%(filename)s:%(lineno)d] - %(message)s',
        datefmt='%Y-%m-%d %H:%M:%S'
    )

    # Console handler
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.INFO)
    console_handler.setFormatter(console_formatter)

    # File handler with rotation
    file_handler = RotatingFileHandler(
        log_file,
        maxBytes=10*1024*1024,  # 10MB
        backupCount=5
    )
    file_handler.setLevel(logging.DEBUG)
    file_handler.setFormatter(file_formatter)

    # Add handlers to logger
    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

```

```python
    return logger

# Usage
logger = setup_logger('ml_pipeline', 'pipeline.log')
logger.info("Logger configured successfully")
logger.debug("This goes to file only")
```

6.3 Configuration Using Dictionary

For complex applications, dictionary-based configuration provides a clean, declarative approach.

```python
import logging
import logging.config

LOGGING_CONFIG = {
    'version': 1,
    'disable_existing_loggers': False,

    'formatters': {
        'standard': {
            'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        },
        'detailed': {
            'format': '%(asctime)s - %(name)s - %(levelname)s - '
                      '[%(filename)s:%(lineno)d] - %(message)s'
        }
    },

    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'INFO',
            'formatter': 'standard',
            'stream': 'ext://sys.stdout'
        },
        'file': {
            'class': 'logging.handlers.RotatingFileHandler',
            'level': 'DEBUG',
            'formatter': 'detailed',
            'filename': 'app.log',
            'maxBytes': 10485760,  # 10MB
            'backupCount': 5
        },
        'error_file': {
            'class': 'logging.FileHandler',
            'level': 'ERROR',
            'formatter': 'detailed',
            'filename': 'errors.log'
        }
    },

    'loggers': {
        '': {  # Root logger
            'handlers': ['console', 'file', 'error_file'],
            'level': 'DEBUG',
            'propagate': False
        },
        'ml_model': {  # Specific logger
            'handlers': ['console', 'file'],
            'level': 'INFO',
            'propagate': False
        }
    }
}

# Apply configuration
logging.config.dictConfig(LOGGING_CONFIG)

# Use loggers
root_logger = logging.getLogger()
ml_logger = logging.getLogger('ml_model')

root_logger.info("Root logger message")
ml_logger.info("ML model logger message")
```

6.4 Configuration from File

You can store configuration in external files for easy management.

6.4.1 YAML Configuration File

Create `logging_config.yaml`:

```yaml
version: 1
disable_existing_loggers: False

formatters:
  standard:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'

handlers:
  console:
    class: logging.StreamHandler
    level: INFO
    formatter: standard
    stream: ext://sys.stdout

  file:
    class: logging.handlers.RotatingFileHandler
    level: DEBUG
    formatter: standard
    filename: app.log
    maxBytes: 10485760
    backupCount: 5

loggers:
  '':
    handlers: [console, file]
    level: DEBUG
    propagate: False
```

Load and apply:

```python
import logging.config
import yaml
```

```

with open(’logging_config.yaml’, ’r’) as f:
    config = yaml.safe_load(f)
    logging.config.dictConfig(config)

logger = logging.getLogger(__name__)
logger.info("Configuration loaded from YAML")
```

7 Best Practices for MLOps

7.1 General Logging Best Practices

1. **Use __name__ for logger names**

   This creates a hierarchical logger structure based on your module organization:

```
# Good practice
logger = logging.getLogger(__name__)

# Avoid
logger = logging.getLogger('my_logger')

```

2. **Use appropriate logging levels**

   - DEBUG: Development and debugging only
   - INFO: Production tracking and milestones
   - WARNING: Recoverable issues and deprecations
   - ERROR: Failures that affect functionality
   - CRITICAL: System-level failures

3. **Use lazy formatting for performance**

```
# Good - lazy evaluation (only formats if logged)
logger.info("Processing batch %d with size %d", batch_id, size)

# Bad - eager evaluation (always formats, even if not logged)
logger.info(f"Processing batch {batch_id} with size {size}")

# Also bad
logger.info("Processing batch " + str(batch_id) + " with size " +
    str(size))

```

4. **Use logger.exception() in except blocks**

```
try:
    model.predict(data)
except Exception as e:
    logger.exception("Prediction failed")
    # Automatically includes full traceback
    raise  # Re-raise after logging

```

5. **Don't log sensitive information**

```
# Bad - security risk
logger.info(f"User password: {password}")
logger.info(f"API key: {api_key}")

# Good - no sensitive data
logger.info(f"User authentication attempt for: {username}")
logger.info("API authentication successful")

```

6. **Configure logging once at startup**

   Set up logging configuration at application entry point, not in every module.

7. **Use structured logging for complex data**

```
# Use extra parameter for structured data
logger.info(
    "Model evaluation complete",
    extra={
        ’accuracy’: 0.95,
        ’precision’: 0.93,
        ’recall’: 0.97,
        ’f1_score’: 0.95
    }
)
```

8. **Avoid excessive logging**

```
# Bad - logs every iteration (too verbose)
for i in range(10000):
    logger.debug(f"Processing item {i}")

# Good - log periodically
for i in range(10000):
    if i % 1000 == 0:
        logger.info(f"Processed {i}/10000 items")
```

7.2 MLOps-Specific Best Practices

7.2.1 Log Model Training Progress

```
import logging

logger = logging.getLogger(__name__)

def train_model(model, train_loader, val_loader, epochs):
    logger.info("=" * 60)
    logger.info("STARTING MODEL TRAINING")
    logger.info("=" * 60)
    logger.info(f"Training samples: {len(train_loader.dataset)}")
    logger.info(f"Validation samples: {len(val_loader.dataset)}")
    logger.info(f"Epochs: {epochs}")

    for epoch in range(epochs):
        # Training phase
        train_loss = train_epoch(model, train_loader)

        # Validation phase
        val_loss, val_acc = validate(model, val_loader)

        # Log progress
        logger.info(
            f"Epoch {epoch+1}/{epochs}: "
            f"train_loss={train_loss:.4f}, "
            f"val_loss={val_loss:.4f}, "
            f"val_acc={val_acc:.4f}"
        )

        # Log warnings if needed
        if val_loss > train_loss * 1.5:
            logger.warning("Possible overfitting detected")

    logger.info("Training completed successfully")
    logger.info(f"Final validation accuracy: {val_acc:.4f}")
    logger.info("=" * 60)
```

7.2.2 Log Data Pipeline Steps

```
def process_data_pipeline(data_source):
    logger.info(f"Loading data from {data_source}")
    data = load_data(data_source)
    logger.debug(f"Raw data shape: {data.shape}")

    logger.info("Starting data preprocessing")
    missing_count = data.isnull().sum().sum()
    if missing_count > 0:
        logger.warning(f"Found {missing_count} missing values")

    cleaned_data = preprocess(data)
    logger.debug(f"Cleaned data shape: {cleaned_data.shape}")

    logger.info("Feature engineering started")
    features = engineer_features(cleaned_data)
    logger.info(f"Created {features.shape[1]} features")
    logger.debug(f"Feature names: {list(features.columns)}")

    logger.info("Data pipeline completed successfully")
    logger.info(f"Final dataset shape: {features.shape}")

    return features
```

7.2.3 Log Model Predictions with Timing

```
import time

def predict_with_logging(model, input_data, request_id=None):
    if request_id is None:
        request_id = str(time.time())

    logger.info(f"[{request_id}] Prediction request received")
    logger.debug(f"[{request_id}] Input shape: {input_data.shape}")

    try:
        start_time = time.time()
        predictions = model.predict(input_data)
        inference_time = time.time() - start_time

        logger.info(
            f"[{request_id}] Prediction completed in "
            f"{inference_time:.3f}s"
        )
        logger.debug(
            f"[{request_id}] Predictions: {predictions[:5]}..."
        )  # Log first 5 only

        return predictions

    except Exception as e:
        logger.exception(f"[{request_id}] Prediction failed")
        raise
```

7.2.4 Log Experiment Configuration

```
def log_experiment_config(config):
    logger.info("=" * 70)
    logger.info("EXPERIMENT CONFIGURATION")
    logger.info("=" * 70)

    for section, params in config.items():
        logger.info(f"{section}:")
        if isinstance(params, dict):
            for key, value in params.items():
                logger.info(f"  {key}: {value}")
        else:
            logger.info(f"  {params}")

    logger.info("=" * 70)

# Usage
config = {
    ’model’: {
        ’type’: ’RandomForest’,
        ’n_estimators’: 100,
        ’max_depth’: 10
    },
    ’training’: {
        ’epochs’: 50,
        ’batch_size’: 32,
        ’learning_rate’: 0.001
    },
    ’data’: {
        ’train_split’: 0.8,
        ’random_state’: 42
    }
}

log_experiment_config(config)
```

7.3 Error Handling Best Practices

```
def load_and_process_data(file_path):
    try:
        logger.info(f"Attempting to load data from {file_path}")
        data = pd.read_csv(file_path)

        logger.info(f"Data loaded successfully: {data.shape}")
        logger.debug(f"Columns: {list(data.columns)}")
```

```python

        # Data validation
        if data.empty:
            logger.error("Loaded data is empty")
            raise ValueError("Empty dataset")

        # Processing
        logger.info("Starting data processing")
        processed_data = process(data)

        logger.info("Data processing completed successfully")
        return processed_data

    except FileNotFoundError:
        logger.error(f"File not found: {file_path}")
        logger.error("Please check the file path and try again")
        raise

    except pd.errors.EmptyDataError:
        logger.error(f"Empty data file: {file_path}")
        raise

    except pd.errors.ParserError as e:
        logger.error(f"Error parsing CSV file: {str(e)}")
        logger.error("Check file format and encoding")
        raise

    except MemoryError:
        logger.critical(
            f"Out of memory while loading {file_path}"
        )
        logger.critical("Try loading data in chunks")
        raise

    except Exception as e:
        logger.exception(
            f"Unexpected error loading data from {file_path}"
        )
        raise
```

8 Complete Real-World Examples

8.1 ML Training Script with Comprehensive Logging

```python
import logging
from logging.handlers import RotatingFileHandler
import time
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score,
    recall_score, f1_score
)

# ============================================
# Logger Configuration
# ============================================

def setup_training_logger():
    """Setup logger for ML training with file rotation"""
    logger = logging.getLogger(__name__)
    logger.setLevel(logging.DEBUG)

    # Console handler - INFO and above
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.INFO)
    console_formatter = logging.Formatter(
        '%(levelname)s - %(message)s'
    )
    console_handler.setFormatter(console_formatter)

    # File handler - all messages with rotation
    file_handler = RotatingFileHandler(
        'training.log',
        maxBytes=10*1024*1024,  # 10MB
        backupCount=5
    )
    file_handler.setLevel(logging.DEBUG)
    file_formatter = logging.Formatter(
        '%(asctime)s - %(name)s - %(levelname)s - '
        '[%(filename)s:%(lineno)d] - %(message)s',
        datefmt='%Y-%m-%d %H:%M:%S'
    )
    file_handler.setFormatter(file_formatter)

    # Add handlers
    logger.addHandler(console_handler)
    logger.addHandler(file_handler)

    return logger

logger = setup_training_logger()

# ============================================
# Training Functions
# ============================================

def validate_data(X, y):
    """Validate input data with logging"""
    logger.debug("Validating input data")

    if X is None or y is None:
        logger.error("Input data is None")
        raise ValueError("Invalid input data")

    if len(X) != len(y):
        logger.error(
            f"Shape mismatch: X has {len(X)} samples, "
            f"y has {len(y)} samples"
        )
        raise ValueError("X and y length mismatch")

    if len(X) < 100:
        logger.warning(
            f"Small dataset: only {len(X)} samples"
        )

    logger.debug(f"Data validation passed: {len(X)} samples")

def train_model(X, y, config):
    """
    Train ML model with comprehensive logging

    Args:
        X: Feature matrix
        y: Target vector
        config: Training configuration dictionary

    Returns:
        Trained model
    """

    logger.info("=" * 70)
    logger.info("STARTING MODEL TRAINING")
    logger.info("=" * 70)

    # Log configuration
    logger.info("Training Configuration:")
    for key, value in config.items():
        logger.info(f"  {key}: {value}")
    logger.info("-" * 70)

    try:
        # Validate data
        validate_data(X, y)

        # Split data
        logger.info("Splitting data into train and test sets")
        X_train, X_test, y_train, y_test = train_test_split(
            X, y,
            test_size=config['test_size'],
            random_state=config['random_state']
        )

        logger.info(f"Training set: {len(X_train)} samples")
        logger.info(f"Test set: {len(X_test)} samples")
        logger.debug(f"Training features shape: {X_train.shape}")
        logger.debug(f"Test features shape: {X_test.shape}")

        # Check class balance
        unique, counts = np.unique(y_train, return_counts=True)
        logger.debug(f"Class distribution: {dict(zip(unique, counts))}")
        if max(counts) / min(counts) > 10:
            logger.warning("Highly imbalanced dataset detected")

        # Initialize model
        logger.info("Initializing Random Forest model")
        logger.debug(f"Model parameters: {config}")

        model = RandomForestClassifier(
            n_estimators=config[’n_estimators’],
            max_depth=config[’max_depth’],
            random_state=config[’random_state’],
            n_jobs=-1,  # Use all CPU cores
            verbose=0
        )

        # Train model
        logger.info("Starting model training...")
        start_time = time.time()

        model.fit(X_train, y_train)

        training_time = time.time() - start_time
        logger.info(
            f"Training completed in {training_time:.2f} seconds"
        )

        # Evaluate on training set
        logger.debug("Evaluating on training set")
        y_train_pred = model.predict(X_train)
        train_acc = accuracy_score(y_train, y_train_pred)
        logger.debug(f"Training accuracy: {train_acc:.4f}")

        # Evaluate on test set
        logger.info("Evaluating model on test set")
        start_eval = time.time()
        y_pred = model.predict(X_test)
        eval_time = time.time() - start_eval

        logger.debug(f"Evaluation time: {eval_time:.3f}s")

        # Calculate metrics
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(
            y_test, y_pred, average=’weighted’, zero_division=0
        )
        recall = recall_score(
            y_test, y_pred, average=’weighted’, zero_division=0
        )
        f1 = f1_score(
            y_test, y_pred, average=’weighted’, zero_division=0
        )

        # Log results
        logger.info("=" * 70)
        logger.info("MODEL EVALUATION RESULTS")
        logger.info("=" * 70)
        logger.info(f"Accuracy:  {accuracy:.4f}")
        logger.info(f"Precision: {precision:.4f}")
        logger.info(f"Recall:    {recall:.4f}")
        logger.info(f"F1-Score:  {f1:.4f}")
        logger.info("=" * 70)

        # Feature importance
        if hasattr(model, 'feature_importances_'):
            importance = model.feature_importances_
            logger.debug(f"Feature importances: {importance}")
            top_features = np.argsort(importance)[-5:][::-1]
            logger.info(
                f"Top 5 important features: {top_features.tolist()}"
            )

        # Performance warnings
        if accuracy < 0.7:
            logger.warning(
                f"Low accuracy: {accuracy:.4f}. "
                "Consider feature engineering or hyperparameter tuning"
            )

        if training_time > 300:  # 5 minutes
            logger.warning(
                f"Long training time: {training_time:.2f}s. "
                "Consider reducing model complexity"
            )

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

        return model

    except ValueError as e:
        logger.error(f"Value error during training: {str(e)}")
        raise

    except MemoryError:
        logger.critical(
            "Out of memory during model training. "
            "Try reducing dataset size or model complexity"
        )
        raise

    except Exception as e:
        logger.exception("Unexpected error during model training")
        raise

# ============================================
# Main Execution
# ============================================

if __name__ == "__main__":
    logger.info("Script started")
    logger.info(f"Python logging version: {logging.__version__}")

    # Configuration
    config = {
        'n_estimators': 100,
        'max_depth': 10,
        'test_size': 0.2,
        'random_state': 42
    }

    try:
        # Generate sample data
        logger.info("Generating sample dataset")
        np.random.seed(42)
        X = np.random.rand(1000, 10)
        y = np.random.randint(0, 2, 1000)
        logger.info("Sample data generated successfully")

        # Train model
        model = train_model(X, y, config)

        logger.info("Script completed successfully")

    except KeyboardInterrupt:
        logger.warning("Script interrupted by user")
    except Exception as e:
        logger.critical("Script failed with critical error")
        logger.exception("Error details")
        raise
```

8.2 Data Pipeline with Logging

```
import logging
import pandas as pd
import numpy as np
from datetime import datetime

logger = logging.getLogger(__name__)

class DataPipeline:
    """
    Data pipeline with comprehensive logging

    Handles data loading, cleaning, and feature engineering
    with detailed logging at each step.
    """

    def __init__(self, config):
        self.config = config
        logger.info("DataPipeline initialized")
        logger.debug(f"Pipeline config: {config}")

    def load_data(self, file_path):
        """Load data with error handling and logging"""
```

```python
        try:
            logger.info(f"Loading data from: {file_path}")
            start_time = datetime.now()

            data = pd.read_csv(file_path)

            load_time = (datetime.now() - start_time).total_seconds()
            logger.info(f"Data loaded in {load_time:.2f} seconds")
            logger.info(f"Dataset shape: {data.shape}")
            logger.debug(f"Columns ({len(data.columns)}): {list(data.columns)}")
            logger.debug(f"Memory usage: {data.memory_usage(deep=True).sum() / 1024**2:.2f} MB")

            # Log data types
            logger.debug("Data types:")
            for col, dtype in data.dtypes.items():
                logger.debug(f"  {col}: {dtype}")

            return data

        except FileNotFoundError:
            logger.error(f"File not found: {file_path}")
            logger.error("Please verify the file path")
            raise

        except pd.errors.ParserError as e:
            logger.error(f"Error parsing CSV: {str(e)}")
            logger.error("Check file format, delimiter, and encoding")
            raise

        except pd.errors.EmptyDataError:
            logger.error(f"Empty file: {file_path}")
            raise

        except Exception as e:
            logger.exception("Unexpected error loading data")
            raise

    def clean_data(self, data):
        """Clean data with detailed logging"""
        logger.info("Starting data cleaning")

        original_shape = data.shape
        original_rows = original_shape[0]
        logger.debug(f"Original shape: {original_shape}")

        # Check missing values
        missing_count = data.isnull().sum().sum()
        if missing_count > 0:
            missing_pct = (missing_count / data.size) * 100
            logger.warning(
                f"Found {missing_count} missing values "
                f"({missing_pct:.2f}% of data)"
            )

            # Log missing values per column
            missing_cols = data.isnull().sum()
            missing_cols = missing_cols[missing_cols > 0]
            for col, count in missing_cols.items():
                col_pct = (count / len(data)) * 100
                logger.debug(
                    f"  {col}: {count} missing ({col_pct:.1f}%)"
                )

            # Handle missing values
            logger.info("Handling missing values")
            numeric_cols = data.select_dtypes(
                include=[np.number]
            ).columns
            data[numeric_cols] = data[numeric_cols].fillna(
                data[numeric_cols].mean()
            )
            logger.info("Numeric missing values filled with mean")

                        categorical_cols = data.select_dtypes(
                include=['object']
            ).columns
            data[categorical_cols] = data[categorical_cols].fillna(
                data[categorical_cols].mode().iloc[0]
            )
            logger.info("Categorical missing values filled with mode
")
        else:
            logger.info("No missing values found")

        # Remove duplicates
        duplicates = data.duplicated().sum()
        if duplicates > 0:
            dup_pct = (duplicates / len(data)) * 100
            logger.warning(
                f"Found {duplicates} duplicate rows ({dup_pct:.2f}%)
"
            )
            data = data.drop_duplicates()
            logger.info(f"Removed {duplicates} duplicate rows")
        else:
            logger.info("No duplicates found")

        # Remove outliers (optional)
        numeric_cols = data.select_dtypes(include=[np.number]).
columns
        outliers_removed = 0
        for col in numeric_cols:
            Q1 = data[col].quantile(0.25)
            Q3 = data[col].quantile(0.75)
            IQR = Q3 - Q1
            lower_bound = Q1 - 1.5 * IQR
            upper_bound = Q3 + 1.5 * IQR

            outliers = ((data[col] < lower_bound) |
                       (data[col] > upper_bound)).sum()
            if outliers > 0:
                logger.debug(f"  {col}: {outliers} outliers detected")
                outliers_removed += outliers

        if outliers_removed > 0:
            logger.info(f"Total outliers detected: {outliers_removed}")

        final_shape = data.shape
        rows_removed = original_rows - final_shape[0]

        logger.info("Data cleaning completed")
        logger.info(f"Final shape: {final_shape}")
        logger.info(f"Rows removed: {rows_removed}")

        return data

    def feature_engineering(self, data):
        """Perform feature engineering with logging"""
        logger.info("Starting feature engineering")

        original_features = data.shape[1]
        logger.debug(f"Original features: {original_features}")

        try:
            # Example: Create interaction features
            logger.debug("Creating derived features")

            # Add timestamp features if date column exists
            date_cols = data.select_dtypes(
                include=['datetime64']
            ).columns
            if len(date_cols) > 0:
                logger.info(
                    f"Processing {len(date_cols)} datetime columns"
                )
                for col in date_cols:
                    data[f'{col}_year'] = data[col].dt.year
                    data[f'{col}_month'] = data[col].dt.month
                    data[f'{col}_day'] = data[col].dt.day
                    logger.debug(
                        f"Created time features from {col}"
                    )

            new_features = data.shape[1]
            added_features = new_features - original_features

            logger.info("Feature engineering completed")
            logger.info(f"Total features: {new_features}")
            logger.info(f"Added features: {added_features}")
            logger.debug(f"New columns: {list(data.columns)}")

            return data

        except Exception as e:
            logger.exception("Error during feature engineering")
            raise

```

```python
    def run(self, file_path):
        """Execute complete pipeline"""
        logger.info("=" * 70)
        logger.info("STARTING DATA PIPELINE")
        logger.info("=" * 70)

        pipeline_start = datetime.now()

        try:
            # Load data
            data = self.load_data(file_path)

            # Clean data
            data = self.clean_data(data)

            # Feature engineering
            data = self.feature_engineering(data)

            pipeline_time = (
                datetime.now() - pipeline_start
            ).total_seconds()

            logger.info("=" * 70)
            logger.info("PIPELINE COMPLETED SUCCESSFULLY")
            logger.info(
                f"Total pipeline time: {pipeline_time:.2f} seconds"
            )
            logger.info("=" * 70)

            return data

        except Exception as e:
            logger.critical("Pipeline failed")
            logger.exception("Pipeline error details")
            raise

# Usage example
if __name__ == "__main__":
    logging.basicConfig(
        level=logging.DEBUG,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.FileHandler('pipeline.log'),
            logging.StreamHandler()
        ]
    )

    config = {'version': '1.0', 'mode': 'production'}
    pipeline = DataPipeline(config)

    # Run pipeline
    try:
        result = pipeline.run('data.csv')
        logger.info("Pipeline execution successful")
    except Exception:
        logger.error("Pipeline execution failed")
```

8.3 Model Serving API with Logging

```python
import logging
from flask import Flask, request, jsonify
import numpy as np
import time
import uuid

app = Flask(__name__)

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('api.log'),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger(__name__)

# Global model variable
model = None

@app.before_first_request
def setup():
    """Initialize application"""
    global model
    logger.info("=" * 70)
    logger.info("INITIALIZING MODEL SERVING API")
    logger.info("=" * 70)

    try:
        logger.info("Loading model")
        # model = load_model('model.pkl')  # Placeholder
        logger.info("Model loaded successfully")
        logger.info("API initialization complete")
    except Exception as e:
        logger.critical("Failed to load model")
        logger.exception("Model loading error")
        raise

@app.route('/predict', methods=['POST'])
def predict():
    """Prediction endpoint with comprehensive logging"""
    request_id = str(uuid.uuid4())[:8]

    logger.info(f"[{request_id}] Prediction request received")
    logger.debug(f"[{request_id}] Request headers: {dict(request.headers)}")

    try:
        # Parse request
        data = request.get_json()
        logger.debug(f"[{request_id}] Request data keys: {list(data.keys())}")

        # Validate request
        if not data or 'features' not in data:
            logger.warning(
                f"[{request_id}] Invalid request: missing 'features'"
            )
            return jsonify({'error': 'Missing features field'}), 400

        features = np.array(data['features'])
        logger.debug(f"[{request_id}] Features shape: {features.shape}")

        # Validate input shape
        if features.ndim != 2:
            logger.error(
                f"[{request_id}] Invalid shape: {features.shape}. "
                f"Expected 2D array"
            )
            return jsonify({
                'error': f'Invalid feature shape: {features.shape}'
            }), 400

        # Check for invalid values
        if np.isnan(features).any():
            logger.error(f"[{request_id}] NaN values in input")
            return jsonify({'error': 'NaN values in features'}), 400

        if np.isinf(features).any():
            logger.error(f"[{request_id}] Inf values in input")
            return jsonify({'error': 'Inf values in features'}), 400

        # Make prediction
        logger.info(f"[{request_id}] Starting prediction")
        start_time = time.time()

        # prediction = model.predict(features)  # Placeholder
        prediction = np.random.rand(features.shape[0])  # Mock prediction

        inference_time = time.time() - start_time

        logger.info(
            f"[{request_id}] Prediction completed in "
            f"{inference_time:.3f}s"
        )
        logger.debug(
            f"[{request_id}] Prediction shape: {prediction.shape}"
        )
        logger.debug(
            f"[{request_id}] Sample predictions: {prediction[:3]}"
        )

        # Log slow predictions
        if inference_time > 1.0:
            logger.warning(
                f"[{request_id}] Slow prediction: {inference_time:.3f}s"
            )

        response = {
            'request_id': request_id,
            'prediction': prediction.tolist(),
            'inference_time': inference_time,
            'status': 'success'
        }

        logger.info(f"[{request_id}] Request completed successfully")
        return jsonify(response), 200

    except ValueError as e:
        logger.error(f"[{request_id}] Value error: {str(e)}")
        return jsonify({
            'error': 'Invalid input values',
            'details': str(e)
        }), 400

    except Exception as e:
        logger.exception(f"[{request_id}] Prediction failed")
        return jsonify({
            'error': 'Internal server error',
            'request_id': request_id
        }), 500

@app.route('/health', methods=['GET'])
def health():
    """Health check endpoint"""
    logger.debug("Health check requested")
    return jsonify({
        'status': 'healthy',
        'timestamp': time.time()
    }), 200

@app.route('/metrics', methods=['GET'])
def metrics():
    """Metrics endpoint"""
    logger.debug("Metrics requested")
    # Return application metrics
    return jsonify({
        'requests_total': 0,  # Placeholder
        'errors_total': 0,    # Placeholder
        'avg_inference_time': 0.0  # Placeholder
    }), 200

if __name__ == '__main__':
    logger.info("Starting Flask application")
    logger.info("Server configuration:")
    logger.info("  Host: 0.0.0.0")
    logger.info("  Port: 5000")
    logger.info("  Debug: False")

    app.run(host='0.0.0.0', port=5000, debug=False)
```

9 Advanced Logging Topics

9.1 Logger Hierarchy

Loggers follow a hierarchical naming structure using dots (.), similar to Python's module structure.

```python
import logging

# Parent logger
parent_logger = logging.getLogger('myapp')

# Child loggers (automatically inherit from parent)
data_logger = logging.getLogger('myapp.data')
model_logger = logging.getLogger('myapp.model')
api_logger = logging.getLogger('myapp.api')

# Grandchild logger
preprocessing_logger = logging.getLogger('myapp.data.preprocessing')
```

Propagation Rules:

- Child loggers inherit settings from parent loggers
- Messages propagate up the hierarchy by default
- Set `propagate=False` to prevent propagation
- Root logger is at the top of all hierarchies

```python
# Configure parent logger
parent = logging.getLogger('myapp')
parent.setLevel(logging.INFO)
handler = logging.StreamHandler()
parent.addHandler(handler)

# Child automatically inherits configuration
child = logging.getLogger('myapp.module')
child.info("This uses parent's handler")  # Works!

# Disable propagation
child.propagate = False  # Now won't use parent's handlers
```

9.2 Custom Log Levels

You can define custom logging levels for specialized needs:

```python
import logging

# Define custom level between DEBUG and INFO
TRACE = 5
logging.addLevelName(TRACE, "TRACE")

def trace(self, message, *args, **kwargs):
    """Add trace method to logger"""
    if self.isEnabledFor(TRACE):
        self._log(TRACE, message, args, **kwargs)

```

```python
# Add method to Logger class
logging.Logger.trace = trace

# Usage
logger = logging.getLogger(__name__)
logger.setLevel(TRACE)

handler = logging.StreamHandler()
handler.setLevel(TRACE)
logger.addHandler(handler)

logger.trace("This is a trace message")  # Works!
logger.debug("This is a debug message")
```

9.3 Filters

Filters provide fine-grained control over which log records are processed:

```python
import logging

class LevelRangeFilter(logging.Filter):
    """Filter to only allow specific level range"""

    def __init__(self, min_level, max_level):
        super().__init__()
        self.min_level = min_level
        self.max_level = max_level

    def filter(self, record):
        return self.min_level <= record.levelno <= self.max_level

# Usage
logger = logging.getLogger(__name__)
handler = logging.FileHandler('info_warnings.log')

# Only log INFO and WARNING (not DEBUG, ERROR, CRITICAL)
level_filter = LevelRangeFilter(logging.INFO, logging.WARNING)
handler.addFilter(level_filter)

logger.addHandler(handler)

logger.debug("Not logged")        # Below min
logger.info("Logged")             # In range
logger.warning("Logged")          # In range
logger.error("Not logged")        # Above max
```

9.3.1 Context Filter Example

```python
class ContextFilter(logging.Filter):
    """Add contextual information to logs"""

    def __init__(self, user_id=None):
        super().__init__()
        self.user_id = user_id

    def filter(self, record):
        record.user_id = self.user_id or 'anonymous'
        return True

# Setup
logger = logging.getLogger(__name__)
handler = logging.StreamHandler()
formatter = logging.Formatter(
    '%(asctime)s - [User:%(user_id)s] - %(levelname)s - %(message)s'
)
handler.setFormatter(formatter)

# Add filter
context_filter = ContextFilter(user_id='user123')
handler.addFilter(context_filter)

logger.addHandler(handler)

logger.info("Processing request")
# Output: 2024-12-17 14:30:45 - [User:user123] - INFO - Processing request
```

9.4 Logging in Multiprocessing

When using multiprocessing, special care is needed to avoid conflicts:

```python
import logging
from logging.handlers import QueueHandler, QueueListener
from multiprocessing import Queue, Process
import time

def worker_process(queue, worker_id):
    """Worker process with queue-based logging"""
    # Configure worker to use queue
    qh = QueueHandler(queue)
    logger = logging.getLogger()
    logger.addHandler(qh)
    logger.setLevel(logging.INFO)

    # Do work with logging
    logger.info(f"Worker {worker_id} started")
    time.sleep(1)
    logger.info(f"Worker {worker_id} processing")
    time.sleep(1)
    logger.info(f"Worker {worker_id} completed")

if __name__ == '__main__':
    # Create queue for log records
    log_queue = Queue()

    # Setup handlers that will process records
    console_handler = logging.StreamHandler()
    file_handler = logging.FileHandler('multiprocess.log')

    formatter = logging.Formatter(
        '%(asctime)s - [PID:%(process)d] - %(levelname)s - %(message)s'
    )
    console_handler.setFormatter(formatter)
    file_handler.setFormatter(formatter)

    # Create listener to process queue
    listener = QueueListener(
        log_queue,
        console_handler,
        file_handler,
        respect_handler_level=True
    )
    listener.start()

    # Create and start worker processes
    processes = []
    for i in range(4):
        p = Process(target=worker_process, args=(log_queue, i))
        p.start()
        processes.append(p)

    # Wait for all workers
    for p in processes:
        p.join()

    # Stop listener
    listener.stop()

    print("All workers completed")
```

9.5 Structured Logging with JSON

For better log parsing and analysis, use JSON format:

```python
import logging
import json
from datetime import datetime

class JsonFormatter(logging.Formatter):
    """Format logs as JSON"""

    def format(self, record):
        log_data = {
            'timestamp': datetime.utcnow().isoformat(),
            'level': record.levelname,
            'logger': record.name,
            'message': record.getMessage(),
            'module': record.module,
            'function': record.funcName,
            'line': record.lineno
        }

        # Add exception info if present
        if record.exc_info:
            log_data['exception'] = self.formatException(record.exc_info)

        # Add extra fields
        if hasattr(record, 'user_id'):
            log_data['user_id'] = record.user_id
        if hasattr(record, 'request_id'):
            log_data['request_id'] = record.request_id

        return json.dumps(log_data)

# Usage
logger = logging.getLogger(__name__)
handler = logging.FileHandler('app.json.log')
handler.setFormatter(JsonFormatter())
logger.addHandler(handler)

logger.info("User logged in", extra={'user_id': 'user123'})
# Output: {"timestamp": "2024-12-17T14:30:45.123456",
#          "level": "INFO", "logger": "__main__",
#          "message": "User logged in", "user_id": "user123", ...}
```

10 Common Issues and Solutions

10.1 Duplicate Log Messages

Problem: Log messages appear multiple times in output.

Cause: Multiple handlers attached to logger or propagation causing duplicates.

Solutions:

```python
# Solution 1: Check if handlers already exist
logger = logging.getLogger(__name__)
if not logger.handlers:
    handler = logging.StreamHandler()
    logger.addHandler(handler)

# Solution 2: Clear existing handlers
logger.handlers = []
logger.addHandler(new_handler)

# Solution 3: Disable propagation
logger.propagate = False

# Solution 4: Use hasHandlers() to check
if not logger.hasHandlers():
    logger.addHandler(handler)
```

10.2 Logs Not Appearing

Problem: Log messages don't appear in expected output.

Causes and Solutions:

1. **Logger level too high**

```python
# Check and set logger level
logger = logging.getLogger(__name__)
print(f"Current level: {logger.level}")
logger.setLevel(logging.DEBUG)

```

2. **Handler level too high**

```python
# Set handler level appropriately
for handler in logger.handlers:
    print(f"Handler level: {handler.level}")
    handler.setLevel(logging.DEBUG)

```

3. **No handlers attached**

```python
# Verify handlers exist
print(f"Handlers: {logger.handlers}")
if not logger.handlers:
    logger.addHandler(logging.StreamHandler())

```

4. **Root logger not configured**

```python
# Configure root logger
logging.basicConfig(level=logging.DEBUG)

```

10.3 basicConfig() Not Working

Problem: `basicConfig()` appears to have no effect.

Cause: `basicConfig()` only works once and only if root logger has no handlers.

Solutions:

```python
# Solution 1: Use force=True (Python 3.8+)
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(levelname)s - %(message)s',
    force=True  # Reconfigure even if already configured
)

# Solution 2: Manually reset root logger
root = logging.getLogger()
for handler in root.handlers[:]:
    root.removeHandler(handler)
logging.basicConfig(level=logging.DEBUG)

# Solution 3: Check if already configured
if not logging.getLogger().hasHandlers():
    logging.basicConfig(level=logging.DEBUG)
```

10.4 File Handler Not Writing

Problem: Log file is not created or not being written to.

Solutions:

1. **Check file permissions and path**

```python
import os

log_file = 'app.log'
log_dir = os.path.dirname(log_file) or '.'

# Check if directory exists
if not os.path.exists(log_dir):
    os.makedirs(log_dir)

# Check write permissions
if not os.access(log_dir, os.W_OK):
    print(f"No write permission for {log_dir}")

```

2. **Ensure proper levels set**

```python
logger.setLevel(logging.DEBUG)
file_handler.setLevel(logging.DEBUG)

```

3. **Force flush after logging**

```python
for handler in logger.handlers:
    handler.flush()

```

10.5 Performance Issues

Problem: Logging causes performance degradation.

Solutions:

1. **Use lazy formatting**

```python
# Good - only formats if logged
logger.debug("Value: %s", expensive_computation())

# Bad - always computes and formats
logger.debug(f"Value: {expensive_computation()}")

```

2. **Increase logging level in production**

```python
# Development
logger.setLevel(logging.DEBUG)

# Production
logger.setLevel(logging.INFO)

```

3. **Use QueueHandler for async logging**

```python
from logging.handlers import QueueHandler
import queue

log_queue = queue.Queue()
queue_handler = QueueHandler(log_queue)
logger.addHandler(queue_handler)

```

11 Quick Reference Guide

11.1 Essential Commands Cheat Sheet

Basic Setup

```python
import logging

# Quick setup
logging.basicConfig(level=logging.INFO)

# Create logger
logger = logging.getLogger(__name__)

# Set level
logger.setLevel(logging.DEBUG)
```

Logging Messages

```python
# Five standard levels
logger.debug("Detailed diagnostic information")
logger.info("Informational message")
logger.warning("Warning message")
logger.error("Error message")
logger.critical("Critical error message")

# Exception logging (includes traceback)
try:
    risky_operation()
except Exception:
    logger.exception("Operation failed")
```

Handlers and Formatters

```
# Create handlers
console_handler = logging.StreamHandler()
file_handler = logging.FileHandler(’app.log’)

# Rotating file handler
from logging.handlers import RotatingFileHandler
rotating_handler = RotatingFileHandler(
    ’app.log’,
    maxBytes=10*1024*1024,  # 10MB
    backupCount=5
)

# Create formatter
formatter = logging.Formatter(
    ’%(asctime)s - %(name)s - %(levelname)s - %(message)s’
)

# Attach formatter to handler
handler.setFormatter(formatter)

# Add handler to logger
logger.addHandler(handler)
```

Complete Setup Example

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

# Console handler
console = logging.StreamHandler()
console.setLevel(logging.INFO)
console.setFormatter(logging.Formatter(’%(levelname)s - %(message)s’))

# File handler
file_h = logging.FileHandler(’app.log’)
file_h.setLevel(logging.DEBUG)
file_h.setFormatter(logging.Formatter(
    ’%(asctime)s - %(name)s - %(levelname)s - %(message)s’
))

logger.addHandler(console)
logger.addHandler(file_h)

logger.info("Logger configured")
```

11.2 Logging Levels Quick Reference

Level	Numeric	Method	When to Use
DEBUG	10	`logger.debug()`	Detailed diagnostic info
INFO	20	`logger.info()`	General informational messages
WARNING	30	`logger.warning()`	Warning messages
ERROR	40	`logger.error()`	Error messages
CRITICAL	50	`logger.critical()`	Critical errors

11.3 Format String Placeholders

Placeholder	Description
`%(name)s`	Logger name
`%(levelname)s`	Log level name
`%(message)s`	Log message
`%(asctime)s`	Timestamp
`%(filename)s`	Source filename
`%(lineno)d`	Line number
`%(funcName)s`	Function name
`%(process)d`	Process ID
`%(thread)d`	Thread ID

11.4 Best Practices Checklist

☐ Use `logging.getLogger(__name__)` for logger names

☐ Configure logging once at application startup

☐ Use appropriate log levels (DEBUG in dev, INFO+ in prod)

☐ Use lazy formatting: `logger.info("User %s", user)`

☐ Use `logger.exception()` in except blocks

☐ Never log sensitive information (passwords, keys)

☐ Implement log rotation for production systems

☐ Add timestamps to all log messages

☐ Include context (request ID, user ID) when relevant

☐ Test logging configuration before deploying

☐ Monitor log file sizes regularly

☐ Use structured logging (JSON) for complex systems

☐ Set up centralized logging for distributed systems

☐ Document logging conventions for your team

12 Glossary

Logger The main interface that applications use to log messages. Loggers are organized hierarchically by name.

Handler Object responsible for dispatching log records to specific destinations (console, file, network, etc.).

Formatter Specifies the layout of log messages, including which attributes to include and how to format them.

Filter Provides fine-grained control over which log records are processed, beyond simple level filtering.

Log Level Severity indicator for log messages (DEBUG=10, INFO=20, WARNING=30, ERROR=40, CRITICAL=50).

Log Record Object containing all information about a single logging event (message, level, timestamp, location, etc.).

Propagation Process by which log messages are passed from child loggers to parent loggers in the hierarchy.

Root Logger The top-level logger in the hierarchy (`logging.getLogger()` with no name), parent to all other loggers.

StreamHandler Handler that sends log output to streams like sys.stdout or sys.stderr (console output).

FileHandler Handler that writes log messages to a file on disk.

RotatingFileHandler Handler that automatically rotates log files when they reach a specified size.

TimedRotatingFileHandler Handler that rotates log files at specified time intervals (hourly, daily, etc.).

basicConfig() Convenience function for simple logging configuration, works only once per program.

Lazy Formatting Technique where string formatting is delayed until the message is actually logged, improving performance.

Structured Logging Practice of logging data in a structured format (e.g., JSON) rather than plain text, making logs easier to parse and analyze.

QueueHandler Handler that sends log records to a queue for asynchronous processing, useful in multiprocessing environments.

QueueListener Receives log records from a queue and dispatches them to configured handlers.

Logger Hierarchy Tree structure of loggers where child loggers inherit configuration from parent loggers.

Exception Logging Special logging that includes full traceback information, typically done with `logger.exception()`.

Handler Level Minimum severity level a handler will output, independent of the logger's level.

Logger Level Minimum severity level a logger will process before passing to handlers.

Context Information Additional data attached to log records (user ID, request ID, session info) for better traceability.

13 Conclusion

Effective logging is a cornerstone of professional software development and absolutely essential for Machine Learning Operations (MLOps). The Python logging module provides a robust, flexible, and production-ready framework for tracking application behavior, debugging issues, and monitoring systems in production environments.

13.1 Key Takeaways

1. **Never Use print() in Production**

   Always use the logging module instead of print() statements. Logging provides:

   - Severity levels for filtering
   - Flexible output destinations
   - Automatic timestamps and context
   - Production-ready features

2. **Choose Appropriate Log Levels**

   Use the right level for each situation:

   - DEBUG for development and detailed diagnostics
   - INFO for production tracking and milestones
   - WARNING for potential issues that don't stop execution
   - ERROR for failures that affect functionality
   - CRITICAL for system-level failures

3. **Configure Logging Properly**

   Set up logging at application startup with:

   - Appropriate handlers (console, file, rotating)
   - Clear, informative formatters
   - Correct levels for each environment
   - Log rotation to manage disk space

4. **Log at the Right Verbosity**

   Balance information and noise:

   - Too much logging creates noise and performance issues
   - Too little logging misses important events
   - Use DEBUG liberally in development
   - Be selective with INFO/WARNING in production

5. **Include Relevant Context**

   Make logs actionable by including:

   - Request IDs for tracing requests
   - User IDs for user-specific issues
   - Timestamps for temporal analysis
   - Function names and line numbers for debugging

6. **Handle Errors Gracefully**

   Always use proper exception logging:

   - Use `logger.exception()` in except blocks
   - Include full tracebacks for debugging
   - Log before re-raising exceptions
   - Don't swallow exceptions silently

7. **Monitor and Rotate Logs**

   In production systems:

   - Implement log rotation to prevent disk full
   - Monitor log file sizes regularly
   - Archive old logs appropriately
   - Set up alerts for critical errors

13.2 MLOps-Specific Recommendations

For machine learning operations, logging serves critical functions:

- **Model Training**: Track epochs, loss, metrics, and training time
- **Data Pipelines**: Log data loading, cleaning, and transformation steps
- **Model Serving**: Record prediction requests, responses, and inference times
- **Performance Monitoring**: Track resource usage (CPU, GPU, memory)
- **Error Tracking**: Identify and debug production failures quickly
- **Audit Trails**: Maintain compliance and reproducibility records
- **Experiment Tracking**: Document configurations and results

13.3 Implementation Strategy

When implementing logging in your projects:

1. **Start Simple**: Begin with `basicConfig()` for prototypes
2. **Expand Gradually**: Add handlers and formatters as needs grow
3. **Establish Standards**: Define logging conventions for your team
4. **Review Regularly**: Analyze logs to improve logging strategy
5. **Integrate Tools**: Connect logging with monitoring and alerting systems
6. **Test Configuration**: Verify logging works before deploying
7. **Document Practices**: Maintain clear documentation for team members

13.4 Common Pitfalls to Avoid

Warning

Avoid These Common Mistakes:

- Using print() instead of logging in production
- Logging sensitive information (passwords, API keys)
- Over-logging in tight loops (performance impact)
- Not rotating log files (disk space issues)
- Using string concatenation instead of lazy formatting
- Forgetting to set appropriate log levels
- Not including enough context in log messages
- Swallowing exceptions without logging
- Configuring logging multiple times
- Not testing logging configuration

13.5 Next Steps

To continue improving your logging practices:

1. **Practice**: Implement logging in personal projects
2. **Experiment**: Try different handlers and formatters
3. **Integrate**: Connect logging with monitoring tools (ELK, Splunk, Datadog)
4. **Learn Advanced Topics**: Explore async logging, custom handlers, filters
5. **Study Production Systems**: Analyze logging in open-source projects
6. **Optimize**: Profile and optimize logging performance
7. **Automate**: Set up automated log analysis and alerting

13.6 Integration with Other Tools

Python logging integrates well with:

- **MLflow**: Experiment tracking and model registry
- **TensorBoard**: Visualization and monitoring
- **ELK Stack**: Elasticsearch, Logstash, Kibana for log analysis
- **Splunk**: Enterprise log management
- **Datadog**: Cloud monitoring and analytics
- **Sentry**: Error tracking and monitoring
- **CloudWatch**: AWS log management
- **Prometheus/Grafana**: Metrics and dashboards

13.7 Final Recommendations

Best Practices Summary

1. Use `logging.getLogger(__name__)` for all loggers
2. Configure logging once at application startup
3. Use appropriate levels: DEBUG < INFO < WARNING < ERROR < CRITICAL
4. Implement log rotation in production
5. Include context information (timestamps, IDs, locations)
6. Use lazy formatting for performance
7. Never log sensitive information
8. Use `logger.exception()` in except blocks
9. Test logging configuration thoroughly
10. Monitor and analyze logs regularly

13.8 Additional Resources

For further learning and reference:

- **Official Documentation**:
  - Python Logging Module: `https://docs.python.org/3/library/logging.html`
  - Logging Cookbook: `https://docs.python.org/3/howto/logging-cookbook.html`
  - Logging HOWTO: `https://docs.python.org/3/howto/logging.html`
- **Tutorials and Guides**:
  - Real Python Logging Guide: `https://realpython.com/python-logging/`
  - Python Logging Best Practices: `https://www.loggly.com/ultimate-guide/python-logging-basics`
- **MLOps Integration**:
  - MLflow Documentation: `https://www.mlflow.org/docs/latest/tracking.html`
  - TensorBoard Logging: `https://www.tensorflow.org/tensorboard`
- **Log Management Tools**:
  - ELK Stack: `https://www.elastic.co/elastic-stack`
  - Splunk: `https://www.splunk.com/`
  - Datadog: `https://www.datadoghq.com/`
  - Sentry: `https://sentry.io/`
- **Books**:
  - ”Logging and Log Management” by Anton Chuvakin
  - ”Python Logging Essentials” by Chetan Giridhar

Remember

Logging is not just about recording events—it's about making your application observable, debuggable, and maintainable.
Good logging practices will:

- Save hours of debugging time
- Enable quick problem resolution in production
- Provide insights into system behavior
- Build confidence in your deployments
- Support compliance and auditing requirements
- Facilitate team collaboration

Invest time in proper logging setup—it pays dividends throughout the application lifecycle!

End of Python Logging Module Complete Reference Guide

"The most effective debugging tool is still careful thought,
coupled with judiciously placed print statements."
— Brian Kernighan