Logging

As your Python programs grow in size and complexity, relying on print() statements to understand what's happening inside your code becomes inefficient and limiting. This is where **logging** comes in.

Logging is a built-in mechanism in Python that allows developers to record events, track application behavior, and troubleshoot problems in a structured and persistent way. Unlike print(), logging supports severity levels, timestamps, file output, and flexible configuration. It's especially valuable in real-world applications where visibility, diagnostics, and monitoring are essential.

In this chapter, you'll learn how to use Python's logging module to create informative, manageable logs. You'll start with the basics—logging simple messages—and progress to configuring logging across multi-module applications. You'll also explore advanced features like log handlers, formatters, and filters that allow you to tailor logging behavior to your project's needs.

By the end of this chapter, you'll understand how to use logging to improve your debugging process, monitor system behavior, and produce more robust, production-ready Python applications.

Chapter Contents

Introduction to Logging in Python

Why Logging is Important in Applications

Logging plays a crucial role in software development, enabling developers to track application behavior, diagnose errors, and maintain system performance. In a small script, debugging with print() statements may seem sufficient. However, as applications grow in complexity, simple print debugging becomes inefficient, unstructured, and impractical.

Logging provides a structured approach to recording events in an application. It allows developers to store messages at various severity levels, track execution flow, and analyze application behavior over time. In production environments, logs serve as an audit trail, helping teams understand issues that may not be easily reproducible.

Consider a web application handling thousands of requests per second. If a critical bug occurs intermittently, developers cannot rely on print() statements to capture and analyze the issue in realtime. Instead, logging enables persistent and organized records of events, which can be reviewed later to pinpoint errors.

Logging is particularly valuable for:

- **Debugging**: Identifying the cause of errors and unexpected behaviors.
- **Monitoring**: Tracking the performance and status of an application.
- **Auditing**: Keeping a historical record of significant events.
- **Security**: Detecting unauthorized access or suspicious activity.
- **Error Handling**: Logging exceptions and failures for better issue resolution.

By implementing logging, developers gain better visibility into application execution and can proactively address potential issues.

Differences Between print() and Logging

Many beginners use print() for debugging, but it quickly becomes limiting in real-world applications. While print() displays output to the console, logging provides a more structured, scalable, and configurable solution.

Limitations of print() Debugging

1. **No Severity Levels**
   print() does not differentiate between debugging messages and critical errors. Developers must manually track different types of messages, leading to inconsistencies.
2. **Lack of Timestamping**
   print() statements do not include timestamps, making it difficult to trace when an event occurred, especially when debugging long-running applications.
3. **No Persistent Storage**
   Console output from print() is ephemeral. Once a program terminates, all messages are lost unless redirected manually to a file. Logging, on the other hand, provides built-in file handling and log rotation mechanisms.
4. **Manual Removal Required**
   print() statements used for debugging must be manually removed before deploying the application, increasing the risk of leaving stray debugging statements in production code. Logging allows developers to control verbosity using log levels without modifying the source code.

Advantages of Logging

1. **Log Levels for Granular Control**
   The logging module categorizes messages into five standard levels:
   - **DEBUG**: Detailed diagnostic messages useful for debugging.
   - **INFO**: General information about the application's operation.
   - **WARNING**: Indications of potential issues that do not stop execution.
   - **ERROR**: Serious problems that affect application functionality.

  - **CRITICAL**: Severe failures that require immediate attention.

With logging, developers can filter messages based on importance, reducing unnecessary noise in production.

2. **Timestamped Entries**
   Logs automatically include timestamps, making it easy to trace when events occurred.
3. **Configurable Output Destinations**
   Logging messages can be sent to various destinations, such as:
   - Console output
   - Log files
   - Remote servers
   - Email alerts
4. **Persistent and Structured Data**
   Unlike print(), logging allows logs to be formatted, structured, and stored persistently for later analysis.

Overview of Python's Built-in Logging Module

Python provides a built-in logging module that offers a flexible and efficient logging system. It supports simple logging for small scripts as well as advanced configurations for large applications.

A basic logging example looks like this:

```
import logging


logging.basicConfig(level=logging.INFO)

logging.info("Application started successfully.")
```

This script initializes logging at the INFO level and outputs:

```
INFO:root:Application started successfully.
```

The logging module provides powerful features, including:

- **Custom Loggers**: Define multiple loggers with different configurations.
- **Handlers**: Direct logs to various destinations (console, files, or external services).
- **Formatters**: Customize log message structure to include timestamps, function names, and line numbers.
- **Filters**: Control which log messages are processed based on conditions.

By mastering Python's logging module, developers can significantly improve debugging, monitoring, and error tracking, ensuring better software quality and maintainability.

Basic Logging Usage

Python's logging module provides a simple yet powerful way to capture and manage log messages. Unlike using print() for debugging, logging allows for structured and categorized messages that can be filtered, formatted, and stored in different locations. This chapter introduces the fundamentals of logging, including how to import the module, log messages, and understand different logging levels.

Importing the Logging Module

The logging module is included in Python's standard library, meaning no additional installation is required. To use it, simply import the module:

```
import logging
```

This allows access to all logging functionalities, including message generation, severity levels, and configuration options.

Logging a Simple Message

To generate a log message, use the logging functions corresponding to different severity levels. The most basic setup logs messages to the console using logging.basicConfig().

Example:

```
import logging


logging.basicConfig(level=logging.INFO)  # Set the logging level
logging.info("This is an informational message.")
```

Output:

```
INFO:root:This is an informational message.
```

By default, logs are displayed in the format:

```
LOG_LEVEL:LOGGER_NAME:MESSAGE
```

Since no custom logger is defined, root is used as the default logger name. The logging level ensures that only messages at or above the specified level are shown.

Default Logging Levels

Python's logging module defines five standard logging levels, each representing a different severity:

1. **DEBUG (10)** – Provides detailed diagnostic information, useful during development.
2. **INFO (20)** – Confirms that an application is working as expected.
3. **WARNING (30)** – Indicates potential issues but does not stop execution.
4. **ERROR (40)** – Reports serious problems that may cause parts of the application to fail.
5. **CRITICAL (50)** – Highlights severe errors that require immediate attention.

By default, the logging system captures messages at WARNING level and above. To include DEBUG and INFO messages, set a lower logging level in basicConfig().

```
Example:
import logging


logging.basicConfig(level=logging.DEBUG)


logging.debug("This is a debug message.")
logging.info("This is an info message.")
logging.warning("This is a warning message.")
logging.error("This is an error message.")
logging.critical("This is a critical message.")
```

Output:

```
DEBUG:root:This is a debug message.
INFO:root:This is an info message.
WARNING:root:This is a warning message.
ERROR:root:This is an error message.
CRITICAL:root:This is a critical message.
```

Changing the level parameter controls which messages are displayed. For example, setting level=logging.ERROR would hide DEBUG, INFO, and WARNING messages, displaying only ERROR and CRITICAL.

Example: Basic Logging in a Script

Here's a complete script demonstrating basic logging usage:

```
import logging


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


def divide(a, b):
    logging.info(f"Dividing {a} by {b}")
    if b == 0:
        logging.error("Attempt to divide by zero")
        return None
    return a / b


result = divide(10, 2)

logging.info(f"Result: {result}")


result = divide(10, 0)

logging.info(f"Result: {result}")
```

Example output:

```
2025-02-07 12:00:01 - INFO - Dividing 10 by 2

2025-02-07 12:00:01 - INFO - Result: 5.0

2025-02-07 12:00:01 - INFO - Dividing 10 by 0

2025-02-07 12:00:01 - ERROR - Attempt to divide by zero

2025-02-07 12:00:01 - INFO - Result: None
```

In this example:

- Logging captures function execution details.
- An error message is logged when attempting to divide by zero.
- The log format includes timestamps, log levels, and messages for better readability.

Basic logging provides a structured way to track an application's execution. By importing logging, setting a logging level, and using appropriate logging functions, developers can replace inefficient print() debugging with a more robust solution.

In the next chapter, we will explore how to configure logging further by customizing output formats, writing logs to files, and using different loggers for better organization.

Configuring Logging

The default configuration of Python's logging module provides basic logging to the console, but in most real-world applications, logging needs to be more structured and customizable. Developers often need to control the verbosity of logs, format messages in a readable way, and store logs in files for later analysis. This chapter explores how to configure logging levels, customize log formats, and direct logs to files instead of the console.

Setting the Logging Level

Logging levels help filter messages based on their importance. By default, Python logs only WARNING and higher-level messages, but this behavior can be adjusted using logging.basicConfig().

To configure a logging level, specify it using the level parameter:

```
import logging

```

```
logging.basicConfig(level=logging.INFO)


logging.debug("This is a debug message.")    # Not shown
logging.info("This is an info message.")     # Shown
logging.warning("This is a warning message.") # Shown
logging.error("This is an error message.")   # Shown
logging.critical("This is a critical message.") # Shown
```

Common Logging Levels

- **DEBUG (10)** – Detailed debugging information.
- **INFO (20)** – General informational messages about program execution.
- **WARNING (30)** – Alerts about potential problems.
- **ERROR (40)** – Indicates failures that need attention.
- **CRITICAL (50)** – Represents severe issues that may cause the application to crash.

If level=logging.DEBUG is set, all messages (including DEBUG) will be logged. Setting level=logging.ERROR would log only ERROR and CRITICAL messages.

Formatting Log Messages

By default, log messages include only the log level and message. However, logs can be formatted to include additional details such as timestamps, filenames, and line numbers for better traceability.

Basic Formatting

A common log format includes:

- **%(asctime)s** – Timestamp of the log entry
- **%(levelname)s** – Log level (INFO, WARNING, etc.)
- **%(message)s** – The actual log message

To apply a custom format:

```
import logging


logging.basicConfig(level=logging.DEBUG, format="%(asctime)s -
%(levelname)s - %(message)s")


logging.info("Application has started.")
logging.warning("Low disk space warning.")
```

Output:

```
2025-02-07 12:00:01 - INFO - Application has started.
2025-02-07 12:00:01 - WARNING - Low disk space warning.
```

Additional Formatting Options

Other useful placeholders include:

- **%(filename)s** – Name of the script generating the log.
- **%(funcName)s** – Function name from which the log was triggered.
- **%(lineno)d** – Line number in the script where the log was recorded.

Example:

```
logging.basicConfig(level=logging.DEBUG, format="%(asctime)s - %(levelname)s - %(filename)s:%(lineno)d - %(message)s")

logging.debug("This is a debug message.")
```

Output:

```
2025-02-07 12:00:02 - DEBUG - myscript.py:10 - This is a debug message.
```

Logging to a File Instead of the Console

In production, logs are typically written to files instead of—or in addition to—the console. This ensures logs persist across sessions and can be analyzed later. To log messages to a file, specify a filename in basicConfig():

```
import logging



logging.basicConfig(filename="app.log", level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")



logging.info("Logging to a file instead of the console.")
```

Running this script creates app.log with the following content:

```
2025-02-07 12:00:03 - INFO - Logging to a file instead of the console.
```

Appending vs. Overwriting Log Files

By default, basicConfig() **appends** logs to an existing file. To overwrite the file each time the script runs, add filemode="w":

```
logging.basicConfig(filename="app.log", filemode="w", level=logging.INFO, format="%(asctime)s - %(levelname)s - %(message)s")
```

Example: Customizing Log Output

Below is a complete example that:

- Logs messages at different levels.
- Formats messages with timestamps, filenames, and line numbers.

- Saves logs to a file instead of the console.

```
import logging


# Configure logging

logging.basicConfig(

    filename="app.log",

    level=logging.DEBUG,

    format="%(asctime)s - %(levelname)s - %(filename)s:%(lineno)d - 
%(message)s"

)


def process_data():

    logging.info("Starting data processing.")

    logging.debug("Loading data from source...")


    try:

        result = 10 / 2  # Change to 10 / 0 to trigger an error

        logging.info(f"Processing result: {result}")

    except ZeroDivisionError:

        logging.error("Division by zero occurred!", exc_info=True)


    logging.warning("Data processing took longer than expected.")

    logging.critical("System is running low on memory!")


# Run the function

process_data()
```

If run, the app.log file will contain logs like:

```
2025-02-07 12:00:04 - INFO - myscript.py:10 - Starting data 
processing.

2025-02-07 12:00:04 - DEBUG - myscript.py:11 - Loading data from 
source...

2025-02-07 12:00:04 - INFO - myscript.py:14 - Processing result: 5.0

2025-02-07 12:00:04 - WARNING - myscript.py:17 - Data processing 
took longer than expected.
```

```
2025-02-07 12:00:04 - CRITICAL - myscript.py:18 - System is running
low on memory!
```

Configuring logging allows developers to fine-tune what information is captured and where it is stored. Setting the logging level controls message verbosity, formatting improves readability, and logging to a file ensures logs persist for debugging and monitoring.

In the next chapter, we will explore logging levels in more detail, explaining how and when to use DEBUG, INFO, WARNING, ERROR, and CRITICAL to write meaningful log messages.

Configuring Logging Across Multiple Modules in a Python Project

When working with a multi-module Python project, logging needs to be configured so that all modules use a consistent logging setup. The best approach is to define a centralized logging configuration that each module can use. This ensures uniform log formatting, prevents duplicate log entries, and allows logs to be directed to multiple destinations (e.g., console, files, external monitoring systems).

Setting Up a Centralized Logging Configuration

Instead of configuring logging separately in each module, a **dedicated logging configuration file or function** should be created. This configuration is then imported and used in all modules.

Creating a Logging Configuration File (logging_config.py)

This file defines the logging setup that all modules will use.

```
import logging


def setup_logging():

    logging.basicConfig(

        level=logging.DEBUG,  # Change to INFO or ERROR for
production

        format="%(asctime)s - %(levelname)s - %(name)s -
%(message)s",

        handlers=[

            logging.FileHandler("app.log"),  # Log to a file

            logging.StreamHandler()          # Log to the console

        ]

    )
```

This function sets up:

- **Log level** (DEBUG, INFO, etc.).
- **Formatted log messages** with timestamps and module names.

- **Multiple handlers** to log to both a file (app.log) and the console.

Using the Centralized Logger in Modules

Each module should **get its own logger instance** instead of using the root logger. This allows logs to be categorized by module, making debugging easier.

Example: module1.py

```
import logging
from logging_config import setup_logging

setup_logging()  # Ensure logging is configured

logger = logging.getLogger(__name__)  # Logger specific to this module

def function_in_module1():
    logger.debug("Debug message from module1")
    logger.info("Info message from module1")
    logger.warning("Warning message from module1")
```

Example: module2.py

```
import logging
from logging_config import setup_logging

setup_logging()

logger = logging.getLogger(__name__)

def function_in_module2():
    logger.error("Error message from module2")
    logger.critical("Critical error in module2")
```

Each module:

- Calls setup_logging() to configure logging (this is safe because basicConfig() only executes once).
- Creates a **module-specific logger** using logging.getLogger(__name__).
- Logs messages at different severity levels.

Running the Application

Example: main.py (Entry Point)

```
from module1 import function_in_module1

from module2 import function_in_module2


function_in_module1()

function_in_module2()
```

When executed, the console output might look like:

```
2025-02-07 12:00:01 - DEBUG - module1 - Debug message from module1

2025-02-07 12:00:01 - INFO - module1 - Info message from module1

2025-02-07 12:00:01 - WARNING - module1 - Warning message from
module1

2025-02-07 12:00:01 - ERROR - module2 - Error message from module2

2025-02-07 12:00:01 - CRITICAL - module2 - Critical error in module2
```

Additionally, these logs are stored in app.log for future reference.

4. Advanced Logging with dictConfig (For Large Projects)

For **more complex logging configurations**, the logging.config.dictConfig() method can be used instead of basicConfig().

Creating logging_config.py with dictConfig

```
import logging.config


LOGGING_CONFIG = {

    "version": 1,

    "disable_existing_loggers": False,

    "formatters": {

        "detailed": {

            "format": "%(asctime)s - %(levelname)s - %(name)s -
%(message)s"

        }

    },

    "handlers": {

        "file": {

            "class": "logging.FileHandler",

            "filename": "app.log",
            "formatter": "detailed",
            "level": "DEBUG"
        },
        "console": {
            "class": "logging.StreamHandler",
            "formatter": "detailed",
            "level": "INFO"
        }
    },
    "loggers": {
        "module1": {
            "handlers": ["file", "console"],
            "level": "DEBUG",
            "propagate": False
        },
        "module2": {
            "handlers": ["file", "console"],
            "level": "ERROR",
            "propagate": False
        }
    }
}

def setup_logging():
    logging.config.dictConfig(LOGGING_CONFIG)
```

Using the Advanced Logger in Modules

```
import logging
from logging_config import setup_logging

setup_logging()

logger = logging.getLogger("module1")
```

```
logger.debug("This is a debug message from module1")
```

This method allows:

- **Module-specific loggers** with different logging levels (module1 logs DEBUG, while module2 logs ERROR).
- **Multiple handlers** for logs (e.g., file and console).
- **More control over logging behavior** (e.g., disabling certain logs in production).

Best Practices for Logging Across Modules

- **Use logging.getLogger(__name__) in each module** to keep logs modular.
- **Centralize logging configuration** in a separate file (logging_config.py).
- **Use multiple handlers** to log to both files and the console.
- **Ensure logs are timestamped and formatted** for readability.
- **Use dictConfig() for complex projects** requiring fine-grained logging control.

Configuring logging across multiple modules ensures a structured approach to logging. By setting up a central logging configuration and using logging.getLogger(__name__) in each module, logs remain consistent and easy to manage. For large-scale applications, dictConfig() provides advanced customization, ensuring better log management and debugging efficiency.

Handlers, Formatters, and Filters

Python's logging module is built to be highly flexible and configurable, allowing you to direct log messages to different destinations, control how they're formatted, and even decide which messages get processed. This is made possible through three core components: handlers, formatters, and filters.

Handlers determine where your log messages go. Instead of just printing to the console, you can send logs to a file, an email address, a remote server, or even multiple places at once. For example, you might want all debug logs written to a file for later review, while only critical issues appear on the console.

Formatters control the layout and content of log messages. By default, a log message might only show the message itself, but you can include other details like the time of the log entry, the severity level, or the name of the logger. A typical format string might look like '%(asctime)s %(levelname)s: %(message)s', which includes the timestamp, the level of the message (like INFO or ERROR), and the message text.

Filters allow fine-grained control over which log messages are processed by a handler. For example, you can write a filter that only allows messages from a certain module, or that includes only WARNING and above messages for one particular output destination.

Let's look at a complete example where we set up a logger that logs messages to both the console and a file. We'll also apply a custom formatter to make the output more readable:

```
import logging

```

```
# Create a logger
logger = logging.getLogger('my_logger')
logger.setLevel(logging.DEBUG)

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

# Set levels for each handler
console_handler.setLevel(logging.WARNING)
file_handler.setLevel(logging.DEBUG)

# Create a formatter
formatter = logging.Formatter('%(asctime)s %(levelname)s:
%(message)s')

# Attach formatter to handlers
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

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

# Example log messages
logger.debug('This is a debug message')
logger.warning('This is a warning')
logger.error('This is an error')
```

In this example, the logger sends all messages of level DEBUG and higher to a file named app.log, while only messages of level WARNING and higher appear in the console. The formatter ensures that both outputs follow the same timestamped format. You can also add filters to either handler if you need to restrict or customize what gets logged even further.

Summary

This chapter introduces the concept of logging in Python, explaining its role in building reliable and maintainable applications. It begins by highlighting the limitations of using print statements for debugging and emphasizes the need for a structured, scalable solution like logging, especially as applications become more complex. Logging provides a systematic way to track the execution of a program, capture errors, monitor performance, and keep historical records of events, all of which are essential in production environments.

The chapter explores the advantages of using the built-in logging module in Python, including its support for different severity levels, automatic timestamping, and the ability to direct logs to various destinations such as the console, files, or external systems. It explains the standard logging levels—DEBUG, INFO, WARNING, ERROR, and CRITICAL—and how developers can configure which messages are shown or stored based on these levels.

As the chapter progresses, it shows how to configure logging output to include useful information like timestamps, filenames, and line numbers. It also covers how to write logs to files, control log formatting, and manage log persistence. This allows developers to replace ad-hoc print statements with configurable logging that's both more powerful and easier to manage in the long term.

The chapter then explains how to set up centralized logging across multiple Python modules. This ensures consistency in log output throughout a project and makes it easier to track down issues by module or component. For larger projects, the chapter introduces advanced configuration techniques using dictConfig, allowing developers to specify detailed logging behavior such as different handlers, formatters, and loggers per module.

Finally, the chapter covers the use of handlers, formatters, and filters to give developers fine-grained control over where logs go, how they are displayed, and which logs are processed. This allows the creation of robust logging systems that can handle various use cases, from local debugging to enterprise-scale monitoring. Overall, the chapter equips readers with the knowledge to use Python's logging system effectively, making their programs more transparent, maintainable, and production-ready.

Exercises

1. **Basic Logging Setup**
   Write a script that imports the logging module and logs a simple informational message such as "Program started."
2. **Log Messages at All Levels**
   Configure the logger to display all messages, and then log one message for each level: DEBUG, INFO, WARNING, ERROR, and CRITICAL.
3. **Customize Log Format**
   Change the log format so that each message includes the timestamp, the level of the message, and the log message itself.
4. **Log to a File**
   Set up logging so that instead of printing to the console, messages are written to a file called app.log.

5. **Logging in a Function with Error Handling**
   Create a function that divides two numbers. Log the input values, and if division by zero is attempted, log an error.

6. **Use a Named Logger**
   Instead of using the root logger, create and use a named logger. Log a message and ensure the logger's name appears in the output.

7. **Logging Across Multiple Modules**
   Create two separate Python files. Set up a logging configuration in one and use it in both modules to ensure consistent logging behavior.

8. **Use Multiple Handlers**
   Set up one handler to write all logs to a file, and another to print only WARNING and higher messages to the console.

9. **Add a Filter to a Logger**
   Create a custom filter that only allows ERROR and CRITICAL messages to be logged by a specific handler.

10. **Use dictConfig for Complex Logging**
    Create a complete logging configuration using dictConfig. Include multiple formatters, handlers, and loggers, and apply it across a multi-module project.