Logging Basics in Python

Introduction

When a Python program runs, you often need to know what is happening inside it.

For example:

- Did the application start successfully?
- Which function is running?
- Did a user request fail?
- Did a file load correctly?
- Did an API request return an error?
- Why did the program stop working?

A beginner might use:

```
print("Something happened")
```

But real applications usually need something more powerful.

Python provides a built in **`logging` module** for recording what happens while a program runs.

The core idea is:

> **Logging gives your application a structured history of important events.**

---

1. What Is Logging?

Logging means recording information about what your program is doing.

For example:

```
import logging

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

Instead of simply printing text, the application creates a **log record**.

A log record can contain:

- What happened
- How important it is
- When it happened
- Which part of the application generated it

This becomes extremely useful when debugging real applications.

---

2. Why Not Just Use `print()`?

You can use:

```
print("User logged in")
```

But `print()` is mainly for displaying something directly to the console.

Logging provides features designed specifically for application monitoring.

For example:

```
import logging

logging.info("User logged in")
logging.warning("Login attempt failed")
logging.error("Database connection failed")
```

You can control:

- Which messages are shown
- How important they are
- Where they are stored
- What information each message contains
- How logs are formatted

---

3. The `logging` Module

Python includes `logging` in its standard library.

You do not need to install it.

Simply write:

```
import logging
```

Then you can create log messages.

---

4. Your First Log Message

A simple example:

```
import logging

logging.warning("Something needs attention")
```

You may see:

```
WARNING:root:Something needs attention
```

The exact formatting can vary depending on the logging configuration.

Notice that the output contains more than just your message.

It includes:

```
WARNING
root
Something needs attention
```

---

5. Log Levels

One of the most important concepts in logging is **log levels**.

Python provides several standard levels:

```
DEBUG
INFO
WARNING
ERROR
CRITICAL
```

They represent different levels of importance.

Think of them as a scale:

DEBUG

INFO

WARNING

ERROR

CRITICAL

---

6. DEBUG

DEBUG is used for detailed information that helps developers understand what the program is doing.

Example:

```
logging.debug("Checking user permissions")
```

Typical uses:

- Variable values
- Function execution
- Detailed program flow
- Database queries during development
- Internal state

Debug messages are usually more detailed than normal application messages.

---

7. INFO

`INFO` represents normal application activity.

Example:

```
logging.info("Application started")
logging.info("User logged in")
logging.info("File uploaded successfully")
```

Use `INFO` when something important happened normally.

---

8. WARNING

`WARNING` indicates something unexpected or potentially problematic, but the application can continue.

Example:

```
logging.warning("API response is taking longer than expected")
```

Other examples:

```
logging.warning("Configuration value is missing; using default")
```

```
logging.warning("Disk space is getting low")
```

A warning does not necessarily mean the application has failed.

---

9. ERROR

`ERROR` means something failed.

Example:

```
logging.error("Could not connect to the database")
```

Another example:

```
logging.error("Failed to process payment")
```

The application may continue running, but a particular operation failed.

---

10. CRITICAL

`CRITICAL` represents a very serious problem.

Example:

```
logging.critical("Application cannot start")
```

This level is appropriate when a major failure threatens the application's ability to continue.

---

11. Comparing Log Levels

Level	Meaning	Example
DEBUG	Detailed developer information	Checking request data
INFO	Normal application activity	Server started
WARNING	Something unexpected	Configuration missing
ERROR	An operation failed	Database query failed
CRITICAL	Severe application failure	Application cannot start

A useful mental model:

```
DEBUG    → What is happening?
INFO     → What normally happened?
WARNING  → Something may be wrong.
ERROR    → Something failed.
CRITICAL → The application is seriously affected.
```

---

12. Logging Methods

Each level has a corresponding method:

```
logging.debug("Debug message")
logging.info("Information message")
```

```
logging.warning("Warning message")
logging.error("Error message")
logging.critical("Critical message")
```

These methods create log records.

---

13. Why `DEBUG` and `INFO` May Not Appear

Consider:

```
import logging

logging.debug("Debug message")
logging.info("Application started")
logging.warning("Warning message")
```

You may only see:

```
WARNING:root:Warning message
```

Why?

Because logging has a **minimum level**.

By default, Python's basic configuration generally shows messages at `WARNING` and above.

So:

```
DEBUG    □
INFO     □
WARNING  □
```

ERROR
CRITICAL

---

14. Setting the Logging Level

You can configure logging using:

```
import logging

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

Now:

```
logging.debug("Debug message")
logging.info("Information message")
logging.warning("Warning message")
logging.error("Error message")
```

can all be displayed.

---

15. basicConfig()

`basicConfig()` provides simple logging configuration.

Example:

```
import logging

logging.basicConfig(
```

```
    level=logging.INFO
)

logging.info("Application started")
logging.warning("Something unexpected happened")
```

The logging level determines which messages are processed.

If you use:

```
level=logging.INFO
```

then:

```
DEBUG    
INFO     
WARNING  
ERROR    
CRITICAL 
```

---

16. Logging Level Hierarchy

The levels have increasing severity:

```
DEBUG
10

INFO
20

WARNING
30
```

```
ERROR
40

CRITICAL
50
```

If you set:

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

then only:

```
ERROR
CRITICAL
```

are normally shown.

This lets you control how much information your application produces.

---

17. Adding Useful Formatting

Logging becomes much more useful when each message includes context.

For example:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(levelname)s:%(message)s"
```

```
)

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

Output:

```
INFO:Application started
```

---

18. Adding the Time

You can include the timestamp:

```
import logging

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

Example:

```
2026-09-25 19:30:15,123 INFO Application started
```

Now you know **when** the event happened.

---

19. Useful Log Format Fields

Some common formatting fields are:

```
%(asctime)s
```

Time of the log record.

`%(levelname)s`

Log level.

`%(message)s`

The actual message.

`%(name)s`

Logger name.

`%(filename)s`

Source filename.

`%(funcName)s`

Function name.

Example:

```
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(name)s | %(message)s"
)
```

---

20. Logging to a File

Logs do not have to remain in the terminal.

You can write them to a file:

```
import logging

logging.basicConfig(
    filename="app.log",
    level=logging.INFO
)

logging.info("Application started")
logging.warning("Something unexpected happened")
```

Python creates:

```
app.log
```

and writes log messages into it.

---

21. File Logging with Formatting

A more useful configuration:

```
import logging

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

```
logging.info("Application started")
logging.error("Database connection failed")
```

The file may contain:

```
2026-09-25 19:30:15,123 | INFO | Application started
2026-09-25 19:30:17,531 | ERROR | Database connection failed
```

This creates a history of application events.

---

22. Logging Variables

You often need to include information in a log message.

For example:

```
username = "Hafsa"

logging.info("User %s logged in", username)
```

Output:

```
INFO:User Hafsa logged in
```

This style is preferred over manually building strings.

Another example:

```
user_id = 42
```

```
logging.info("Processing user %s", user_id)
```

---

23. Multiple Values

You can provide multiple values:

```
username = "Hafsa"
action = "download"

logging.info(
    "User %s performed %s",
    username,
    action
)
```

Output:

```
INFO:User Hafsa performed download
```

---

24. Do Not Log Sensitive Information

Be careful with logs.

Avoid:

```
logging.info("API key: %s", api_key)
```

Avoid:

```
logging.info("Password: %s", password)
```

Avoid logging sensitive:

- Passwords
- API keys
- Access tokens
- Private authentication data
- Credit card information
- Sensitive personal information

Logs can be stored, copied, shared, or accessed by other systems.

---

25. Logging Exceptions

Logging becomes especially useful when handling errors.

Example:

```
import logging

try:
    result = 10 / 0
except ZeroDivisionError:
    logging.error("Division failed")
```

This records that something went wrong.

But there is an even better option when debugging exceptions.

---

26. logging.exception()

Inside an `except` block, you can use:

```
import logging

try:
    result = 10 / 0
except ZeroDivisionError:
    logging.exception("Calculation failed")
```

This records the error message **and traceback**.

A traceback shows where the exception occurred.

This is extremely useful for debugging.

---

27. Logging an Exception with Context

Example:

```
import logging

filename = "data.txt"

try:
    with open(filename, "r") as file:
        data = file.read()
except FileNotFoundError:
    logging.exception("Could not read file %s", filename)
```

The log now contains useful context and the traceback.

---

28. `logging.error()` vs `logging.exception()`

Use:

```
logging.error("Something failed")
```

when you simply want to record an error.

Use:

```
logging.exception("Something failed")
```

inside an `except` block when you want the traceback as well.

Example:

```
try:
    process_data()
except Exception:
    logging.exception("Data processing failed")
```

---

29. Creating a Logger

For larger applications, instead of using the root `logging` object everywhere, create a logger:

```
import logging
```

`logger = logging.getLogger(__name__)`

Then:

```
logger.info("Application started")
logger.warning("Configuration is missing")
logger.error("Database connection failed")
```

This is the preferred pattern for larger projects.

---

30. Why `__name__`?

Suppose your file is:

`users.py`

Then:

`logger = logging.getLogger(__name__)`

creates a logger whose name corresponds to the module.

If `users.py` is imported as a module, its logger can be identified as:

`users`

This becomes useful in larger applications because you can see **which module produced a log message**.

---

31. Example with a Module Logger

```
import logging

logger = logging.getLogger(__name__)

def login(username):
    logger.info("Login attempt for user %s", username)

    if not username:
        logger.warning("Login attempted without username")
        return False

    return True
```

This is more scalable than using `logging.info()` everywhere.

---

32. Logger Names

Imagine a project:

```
project/
□
□□□ app.py
□□□ users.py
□□□ database.py
□□□ payments.py
```

Each module can have:

```
logger = logging.getLogger(__name__)
```

Then logs can identify their source:

```
INFO users Login attempt
INFO database Connection established
ERROR payments Payment failed
```

This becomes very useful when debugging larger systems.

---

33. A Simple Real World Example

Consider a file processor:

```
import logging

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

logger = logging.getLogger(__name__)

def process_file(filename):
    logger.info("Processing file %s", filename)

    try:
        with open(filename, "r") as file:
            data = file.read()

        logger.info("File processed successfully")
        return data

    except FileNotFoundError:
        logger.error("File not found: %s", filename)
        return None
```

Now the program records meaningful events.

---

34. Logging vs Debugging

Logging and debugging are related but different.

Logging

Records what happens while the application runs.

```
Application started
User logged in
Database connected
Request failed
```

Debugging

Investigates why the program behaves incorrectly.

You may use:

- Logs
- Breakpoints
- Debuggers
- Stack traces
- Tests
- Inspecting variables

Logs are therefore one of the tools developers use while debugging.

---

35. Logging vs `print()`

`print()`	Logging
Simple output	Structured application events
No severity levels	Multiple severity levels
Limited configuration	Highly configurable
Mainly console output	Console, files, and other destinations
Not designed for monitoring	Designed for application monitoring
Harder to control in large apps	Easy to filter by level

A quick script may use `print()`.

A real application generally benefits from logging.

---

36. Logging in a Project

A simple structure:

```
my_app/
□
□□□ app.py
□□□ users.py
□□□ database.py
□□□ logs/
□   □□□ app.log
□□□ requirements.txt
```

app.py:

```
import logging

logging.basicConfig(
    filename="logs/app.log",
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(name)s | %(message)s"
)
```

users.py:

```
import logging

logger = logging.getLogger(__name__)

def create_user(username):
    logger.info("Creating user %s", username)
```

Now different modules can write to the same logging system.

---

37. Common Mistakes

Mistake 1: Logging everything at `ERROR`

Do not write:

```
logging.error("User logged in")
```

A successful login is normally not an error.

Use:

```
logging.info("User logged in")
```

Mistake 2: Using `DEBUG` for critical failures

Do not write:

```
logging.debug("Database is unavailable")
```

Use:

```
logging.error("Database is unavailable")
```

Mistake 3: Logging sensitive data

Never casually log:

```
logging.info("Password: %s", password)
```

Mistake 4: Using vague messages

Avoid:

```
logging.error("Failed")
```

Prefer:

```
logging.error("Failed to load user profile for user %s", user_id)
```

The second message provides context.

Mistake 5: Logging without useful context

Instead of:

```
logging.info("Processing")
```

use:

```
logging.info("Processing resource %s", resource_id)
```

Mistake 6: Replacing every `print()` immediately

Not every tiny script needs a complex logging setup.

Use the appropriate tool for the application.

---

38. A Practical Logging Strategy

When adding logs to an application, ask:

Step 1: What events matter?

Examples:

```
Application startup
User login
File upload
Database connection
API request
Payment failure
```

Step 2: How important is each event?

Choose:

```
DEBUG
INFO
WARNING
ERROR
CRITICAL
```

Step 3: What context is useful?

Include information such as:

```
User ID
File name
Request ID
Operation
Resource ID
```

but avoid sensitive information.

Step 4: Where should logs go?

For simple development:

`Console`

For persistent application logs:

`File`

Production systems may use centralized logging platforms.

---

39. Real World Example: haas.dev

Imagine a Python backend that manages haas.dev resources.

A resource request might produce:

```
import logging

logger = logging.getLogger(__name__)

def get_resource(resource_id):
    logger.info("Fetching resource %s", resource_id)

    try:
        resource = database.get(resource_id)

        if resource is None:
```

```
            logger.warning(
                "Resource %s was not found",
                resource_id
            )
            return None

        logger.info(
            "Resource %s loaded successfully",
            resource_id
        )

        return resource

    except Exception:
        logger.exception(
            "Failed to fetch resource %s",
            resource_id
        )
        return None
```

Now the logs can tell you:

```
INFO    Fetching resource pkfhp6
INFO    Resource pkfhp6 loaded successfully
```

or:

```
INFO    Fetching resource unknown
WARNING Resource unknown was not found
```

or:

```
ERROR   Failed to fetch resource pkfhp6
```

This is much more useful than random `print()` statements.

---

40. Mini Project: Application Logger

Build a small application logger.

Requirements:

1. Configure logging.
2. Show `INFO` and higher messages.
3. Include timestamps.
4. Include the log level.
5. Save logs to `app.log`.
6. Create a logger using `__name__`.
7. Log application startup.
8. Log a successful operation.
9. Log a warning.
10. Log an exception with a traceback.

Example:

```
import logging

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

logger = logging.getLogger(__name__)
```

```
logger.info("Application started")

try:
    result = 10 / 0
except ZeroDivisionError:
    logger.exception("Calculation failed")
```

---

41. Five Minute Challenge

Create a program called:

`resource_processor.py`

The program should:

1. Configure logging.
2. Create a logger with `__name__`.
3. Log when processing starts.
4. Try to read a file.
5. Log successful reading with `INFO`.
6. Log `FileNotFoundError` with `ERROR`.
7. Use `logger.exception()` for unexpected exceptions.
8. Write logs to `resource_processor.log`.

Do not log the complete contents of the file.

---

42. Exercises

Exercise 1

Create five log messages:

`DEBUG`
`INFO`
`WARNING`
`ERROR`
`CRITICAL`

Exercise 2

Configure logging so that only `INFO` and higher messages appear.

Exercise 3

Create a format containing:

`timestamp`
`log level`
`message`

Exercise 4

Write logs to:

`app.log`

Exercise 5

Create:

`logger = logging.getLogger(__name__)`

and use it in a function.

Exercise 6

Create a file-reading function that logs:

- File processing started
- File successfully read
- File not found
- Unexpected exception

---

43. Mini Quiz

1. Which module provides Python's built-in logging system?

A. `log`
B. `logger`
C. `logging`
D. `debug`

Answer: C

---

2. Which level represents normal application activity?

A. ERROR
B. INFO
C. CRITICAL
D. DEBUG

Answer: B

---

3. Which level indicates a potentially problematic situation?

A. WARNING
B. INFO
C. DEBUG
D. CRITICAL

Answer: A

---

4. Which function is useful for logging an exception with its traceback?

A. logging.print()
B. logging.error()
C. logging.exception()
D. logging.trace()

Answer: C

---

5. Why use `logging.getLogger(__name__)`?

A. To create a database
B. To identify logs by module
C. To remove errors
D. To create environment variables

Answer: B

---

44. Interview Questions

Beginner

1. What is logging?

Logging is the process of recording important events and information while an application runs.

2. Why is logging better than `print()` for applications?

Logging provides levels, formatting, filtering, multiple destinations, and structured application diagnostics.

3. What are the standard Python logging levels?

`DEBUG`
`INFO`
`WARNING`
`ERROR`
`CRITICAL`

4. How do you create a basic log message?

```
import logging

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

Intermediate

5. What does `basicConfig()` do?

It provides basic configuration for Python's logging system, such as log level, output file, and message format.

6. What does the logging level control?

It determines which log messages are processed and displayed.

7. What is `logging.exception()` used for?

It is used inside an exception handler to record an error together with its traceback.

8. Why use `getLogger(__name__)`?

It creates a module-specific logger, making logs easier to identify and manage in larger applications.

Advanced

9. Why should sensitive data not be logged?

Because logs can be stored, transmitted, or accessed by people and systems that should not have access to secrets.

10. What makes a good log message?

It should explain what happened, provide useful context, use the appropriate severity level, and avoid sensitive information.

---

45. Cheat Sheet

Import

```
import logging
```

Basic log

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

Levels

```
logging.debug("Debug")
logging.info("Info")
logging.warning("Warning")
logging.error("Error")
logging.critical("Critical")
```

Configure

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

Configure format

```
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(message)s"
```

```
)
```

Log to file

```
logging.basicConfig(
    filename="app.log",
    level=logging.INFO
)
```

Create module logger

```
logger = logging.getLogger(__name__)
```

Use logger

```
logger.info("Something happened")
```

Log variables

```
logger.info("User %s logged in", username)
```

Log exception with traceback

```
try:
    risky_operation()
except Exception:
    logger.exception("Operation failed")
```

---

46. Key Takeaways

- Logging records what happens inside an application.
- Python provides logging through the built-in `logging` module.
- `print()` is useful for simple output, but logging is designed for application diagnostics.
- Python has five commonly used log levels:

DEBUG, INFO, WARNING, ERROR, and CRITICAL.
- basicConfig() provides simple logging configuration.
- Logs can be sent to the console or a file.
- getLogger(__name__) creates a module-specific logger.
- logging.exception() records an exception with its traceback.
- Good logs contain useful context.
- Never log passwords, API keys, tokens, or other sensitive information.
- Use the appropriate severity level instead of marking every event as an error.
- Logging becomes increasingly valuable as applications become larger and harder to debug.

The core idea:

```
print() → Show something

logging → Record what happened
          ↓
      level + time + context
          ↓
      easier debugging
          ↓
      easier monitoring
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app