Sentrial Blog

Sign Up

Get Started Quickly With Python Logging

Jack Cheng
Jan 20, 2022

Writing code that includes logging statements is one of the most popular practices in DevOps today. With a well-thought-out logging in place, developers can easily monitor the health of their software.

Logging is a very simple concept. It is just a matter of writing a few lines of code to record some pieces of information. But there are many subtleties that can make logging very difficult. Some of the questions you need to answer are:

- Which logging levels should I use?
- What should I log?
- Where should I write my logging?
- Should I use a logging framework?

In this post, we will explore Python logging, and we will understand what are its pros and cons.

What's the Motivation for Logging? Why Bother?

Without a doubt, logging is the most standard debugging tool example. If you don't have a clue why a piece of code is not working as you wanted, you can add a few lines of logging statements.

Is there some software factor that might cause an application to fail?

Logging is a very simple and cheap way to debug such a malfunction, and it is effective in:

- Receiving alerts when something unexpected happens
- Review logs to understand what the system is doing (root cause analysis)
- Monitor the health of your software
- Provide a trail for security auditing
- Provide a trail for business analytics
- Provide a trail for customer support (e.g. customer complaints about a specific transaction)

What You Should (and Shouldn't) Log

Logging is a very simple concept, but there are many subtleties that can make it very difficult. In this section, we will review some important practices that you should follow when you write logging statements.

Treat your logs as if they're one day become public.

Which is another reason why we should be very careful when we write logs, we should avoid logging:

- Personal information
- Credit card numbers
- Passwords
- Auth tokens
- Support identity
- Internal system details
- Intellectual property
- Vulnerabilities
- Trade secrets
- Business secrets
- Sensitive business transactions
- Information about vulnerabilities in your system, or ways to circumvent the security of your system

This is a very important rule, and it is very easy to violate it. For example, a developer may want to log a request object for debugging purposes, but this request object may contain sensitive information such as passwords, credit card numbers, etc.

The Simplest Python Logging That Could Work

The simplest logging that we can do is to write a few lines of code to log messages to the console. For example, we can use the print function:

Another option is to use the standard library module logging, which provides more flexibility. For example, we can write:

Logging

Logging is a very simple and powerful way to debug your software. It is very easy to write a few lines of code to log messages to console.

Logging is very flexible. We can write logs to console, to a file, to a remote logging service, etc. We can also use a logging framework to manage our logging.

Logging is very fast. We can write logs to console and to a file without worry about performance issues.

Logging is very secure. We can write logs to console and to a file without worry about sensitive information being exposed.

Logging is very powerful. We can use logging to monitor our software, to debug our code, to do root cause analysis and to support our customers.

Logging is very easy to learn. We can use logging to write our own logs, or we can use a logging framework to manage our logging.

Configuring the Logger

Logging is very flexible. It can be configured to write messages to console, to a file, or to a remote logging service.

- Logger: A logger is the entry point into the logging system. It exposes the API that developers use to log messages in their code.
- Handler: A handler is responsible for sending log messages to the appropriate destination.
- Formatter: A formatter is responsible for formatting log messages into a specified format. It allows developers to add context information to log messages.

Finally, logging handlers can be set up to filter log messages based on their importance. For example, we can configure a handler to only write warnings and errors to a log file, but we can also configure another handler to write all log messages to console.

Configuring Logging in the Code

Logging can be configured in the code. For example, we can use basicConfig() method to set a logging handler to write a log file in logging.log, and we can write:

Configuring Logging in the Log Files

Logging can also be configured in the log files. For example, we can create a file logging.conf, and we can write:

Logging Levels

Logging levels are very important. We can use logging levels to filter out log messages that are not important enough. For example, we can configure a logger to only write warnings and errors to a log file, but we can also configure another logger to write all log messages to console.

Logging levels are hierarchical:

Logging levels are hierarchical. For example, if we set the logging level to WARNING, we will see WARNING, ERROR, and CRITICAL messages, but we will not see INFO or DEBUG messages.

Formatting Log Messages

Formatting log messages is very important. We want to know when, where, and by what code a log message was written.

Logging is very flexible. We can write custom formatters to add information that is not in the log message, for example:

Here is a list of some of the common attributes that we can use:

- %(asctime)s
- 2021-01-18 20:29:30,317
- %(name)s: the name of the logger that generated the log message (by default, this is the root logger, unless we specify a logger)
- %(levelname)s
- WARNING: the level of the logger, this can be a logger name or one of the default logger levels which are (from the most to the lowest):
  - CRITICAL
  - FATAL
  - ERROR
  - WARNING
  - INFO
  - DEBUG
  - NOTSET
- %(filename)s: the name of the file where the logging statement was written
- %(lineno)s: the line number where the logging statement was written
- %(funcName)s: the function where the logging statement was written
- %(message)s: the log message

Note What?

Logging is very powerful and very easy to use. But it is not the only way to do logging. Sometimes, we can use logging frameworks to manage our logging. Logging frameworks are a powerful and necessary tool.

Stay tuned, in our next blog post we will cover logging in Python using Sentrial, a cloud-native, AI-native logging and observability platform.

Logging frameworks are a powerful and necessary tool.

Sentrial is the AI-native platform for AI Observability, AI Evaluations, and AI Agent Control. Check out Sentrial's docs and sign up for a free account to start monitoring your AI agents in production.

Sentrial's Agent Control Plane lets you define rules that detect and control your AI agents' behavior in production, including PII leakage, prompt injection, tool misuse, and custom business logic. Learn more about Agent Control.

Sentrial is SOC 2 Type II compliant and HIPAA-ready, with customer data never used for model training. Pricing starts at $0.01 per trace, with a free tier for prototyping and production workloads. Learn more at sentrial.com.

Also on Sentrial Blog: Evaluating Tool Calling Agents in Production, Agent Drift: Why Your AI Agent Stops Working Consistently

Read more about Agent Control:

- Introducing Agent Control: Control Planes for AI Agents

© 2026 Sentrial. All rights reserved.
Terms of Service · Privacy Policy · Cookie Policy · Cookie Preferences · Do Not Sell or Share My Personal Information