Requirements Document: Logger Class

1. Purpose

Replace all print() calls in the project with a centralized logger that:

- Controls verbosity by severity level.
- Outputs consistent, timestamped log messages.
- Can be configured globally (so you can quiet or expand logs without editing every script).
- Optionally writes logs to file for debugging/backtesting audit trails.

2. Logging Levels

The logger must support at least these levels:

1. DEBUG - Detailed diagnostic information for development (previously printed with print()).
2. INFO - General application events, e.g., "Order submitted", "Fetched 20 studies."
3. WARNING - Something unexpected happened, but the program can continue.
4. ERROR - Recoverable error, likely needs investigation.
5. CRITICAL - Severe error; the application may not be able to continue.

Only messages at or above the configured log level should be printed.

3. Features

3.1 Basic Requirements

- Global Log Level: Configurable at runtime or via constructor (e.g., Logger(level="INFO")).
- Timestamped Messages: Format: YYYY-MM-DD HH:MM:SS [LEVEL] message
- Output Destinations: Always console, optional log file (append mode)
- Singleton-like Behavior: Only one logger instance should be used across the application.

3.2 Convenience Methods

Provide methods:

```
logger.debug(msg)
logger.info(msg)
logger.warning(msg)
logger.error(msg)
logger.critical(msg)
```

3.3 Additional Features (Stretch Goals)

- Color-coded output (DEBUG gray, INFO white, WARNING yellow, ERROR red).
- Configurable modules (different levels per module).
- Async-safe (thread/process safe logging).
- Log rotation (rotate log files daily or after size limit).

4. Example Usage

Initialization:

```
from logger import Logger
logger = Logger(level="INFO", log_to_file=True, file_path="project.log")
```

Usage in code:

```
logger.debug("Starting API request with params ...")
logger.info(f"Fetched {len(studies)} studies")
logger.warning("Primary completion date missing")
logger.error("Failed to connect to Alpaca API")
logger.critical("Unhandled exception in trading loop!")
```

Behavior:

If the log level is set to INFO, only INFO, WARNING, ERROR, CRITICAL messages are shown. DEBUG messages are suppressed.

5. Constraints and Considerations

- Must not significantly degrade runtime performance (avoid excessive I/O).
- Must be easy to switch between verbose debugging and quiet production mode.
- Should work across your scraping, ML, and trading modules.

6. Alternatives Considered

- Use Python's built-in logging module:
  Pros: Stable, feature-rich, thread-safe
  Cons: Slightly verbose to set up; custom class gives simpler syntax
- External libraries (loguru):

Pros: Very clean syntax, rotating logs built-in

Cons: Adds an external dependency

Given your project complexity, you could wrap the standard logging module inside a Logger class that enforces your desired defaults.