SQLite: A Lightweight Database

What is SQLite?

- SQLite is a C library that provides a lightweight, self-contained, serverless, and zero-configuration database engine.
- It stores the entire database in a single file and is commonly used for small-to-medium-scale applications.

Key Features:

- **Lightweight and Portable**: No external server or setup required.
- **SQL Syntax Support**: Provides full SQL syntax for managing and querying data.
- **ACID Compliance**: Ensures reliability in transaction handling.
- **Cross-Platform**: Works seamlessly across different operating systems.

Common SQLite Operations:

Connecting to a Database

python

Copy code

```
import sqlite3


# Connect to SQLite database (creates the file if it doesn't exist)
conn = sqlite3.connect("example.db")
cursor = conn.cursor()
```

Creating a Table

python

Copy code

```
cursor.execute('''
  CREATE TABLE IF NOT EXISTS Users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER NOT NULL
  )
```

''')

conn.commit()

Inserting Data

python

Copy code

cursor.execute("INSERT INTO Users (name, age) VALUES (?, ?)", ("Alice", 30))

conn.commit()

Querying Data

python

Copy code

cursor.execute("SELECT * FROM Users")

for row in cursor.fetchall():

  print(row)

Closing the Connection

python

Copy code

conn.close()

Logging: Capturing Application Events

What is Logging?

- Logging is the practice of recording events or messages during the execution of a program.
- Useful for debugging, monitoring, and auditing applications.

Python logging Module

- The logging module provides a flexible framework for logging messages in Python applications.

Logging Levels:

1. **DEBUG**: Detailed information for diagnostics.
2. **INFO**: General operational messages.

3. **WARNING**: Something unexpected, but the program continues.
4. **ERROR**: Due to a serious issue, the program might fail.
5. **CRITICAL**: A very serious issue; the program might crash.

Basic Usage:

python

Copy code

```
import logging


# Configure logging
logging.basicConfig(level=logging.DEBUG, filename="app.log", filemode="a",
                    format="%(asctime)s - %(levelname)s - %(message)s")


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")
```

Advanced Features:

- **Log Rotation**: Use logging.handlers to rotate logs when they reach a certain size.
- **Custom Handlers**: Stream logs to files, consoles, or even email.

Example with Log Rotation:

python

Copy code

```
from logging.handlers import RotatingFileHandler


logger = logging.getLogger()
handler = RotatingFileHandler("app.log", maxBytes=2000, backupCount=5)
logger.addHandler(handler)
```

logger.warning("This is a warning with rotation")

ConfigParser: Configuration File Management

What is ConfigParser?

- ConfigParser is a Python module for working with configuration files.
- Stores settings and parameters in .ini format.

Structure of .ini Files:

ini

Copy code

[Settings]

debug = True

log_level = DEBUG

db_name = example.db

Basic Usage:

python

Copy code

import configparser

Read configuration

config = configparser.ConfigParser()

config.read("config.ini")

Access sections and keys

debug_mode = config.getboolean("Settings", "debug")

log_level = config.get("Settings", "log_level")

db_name = config.get("Settings", "db_name")

print(debug_mode, log_level, db_name)

Writing Configuration:

python

Copy code

```
config = configparser.ConfigParser()
config["Settings"] = {
   "debug": "True",
   "log_level": "DEBUG",
   "db_name": "example.db"
}

with open("config.ini", "w") as configfile:
   config.write(configfile)
```

Advanced Features:

- **Default Values**:

python

Copy code

```
config["DEFAULT"] = {
   "retries": "3",
   "timeout": "30"
}
```

- **Interpolation**: Referencing other keys within the same file:

ini

Copy code

```
[Paths]
home_dir = /usr/local
logs = %(home_dir)s/logs
```

Integrating SQLite, Logging, and ConfigParser

Comprehensive Example:

python

Copy code

```
import sqlite3
import logging
import configparser

# Read configurations
config = configparser.ConfigParser()
config.read("config.ini")

# Configure logging
logging.basicConfig(level=config.get("Settings", "log_level"), filename="app.log",
                    format="%(asctime)s - %(levelname)s - %(message)s")

# Connect to SQLite database
conn = sqlite3.connect(config.get("Settings", "db_name"))
cursor = conn.cursor()

# Create a sample table
cursor.execute('''
   CREATE TABLE IF NOT EXISTS Logs (
      id INTEGER PRIMARY KEY AUTOINCREMENT,
      message TEXT NOT NULL,
      level TEXT NOT NULL,
      timestamp TEXT NOT NULL
   )
''')
```

```
conn.commit()


# Insert log data into the database
logging.info("Application started")
cursor.execute("INSERT INTO Logs (message, level, timestamp) VALUES (?, ?, datetime('now'))",
        ("Application started", "INFO"))
conn.commit()


# Query logs
cursor.execute("SELECT * FROM Logs")
for row in cursor.fetchall():
   print(row)


conn.close()
logging.info("Application ended")
```

Key Takeaways:

1. **SQLite**: Ideal for lightweight, embedded database solutions.
2. **Logging**: Critical for monitoring and debugging applications effectively.
3. **ConfigParser**: Simplifies managing application configurations in .ini files.

This integration showcases practical usage of all three concepts, enabling efficient application management with logging, configuration, and database operations.