Mastering Python Decorators for Database Operations

Prepared for ALX Backend Python Project

May 20, 2025

Abstract

This comprehensive guide explores Python decorators in the context of database operations, focusing on creating reusable, efficient, and robust code for managing SQLite databases. Designed for novice learners, it covers logging queries, connection management, transaction handling, retry mechanisms, and query caching. Each concept is explained in detail with senior-level code, external references, and practical examples. The guide also includes advanced techniques and best practices for professional backend development.

Contents

1 Introduction to Python Decorators

1.1 What Are Decorators?

A *decorator* in Python is a higher-order function or callable that wraps another function to extend its behavior without modifying its source code. Think of a decorator as a reusable layer, like adding a protective case to a phoneit enhances functionality (e.g., logging, error handling) without altering the core device (the function).

For example, consider a function that fetches data from a database. A decorator can log the query, manage the database connection, or cache the result, making the code cleaner and more maintainable. In this project, we use decorators to automate repetitive database tasks, simulating real-world backend scenarios.

References:

- Video: Corey Schafer: Python Decorators in 15 Minutes
- Article: Real Python: Python Decorators 101
- Book: *Fluent Python* by Luciano Ramalho, Chapter 7
- Documentation: PEP 318 Decorators

1.2 Functions as First-Class Objects

Python treats functions as *first-class objects*, meaning they can be assigned to variables, passed as arguments, or returned from other functions. This is the foundation of decorators.

```
def greet():
    print("Hello!")

my_func = greet  # Assign to variable
my_func()  # Outputs: Hello!

def execute_func(func):
    func()

execute_func(greet)  # Outputs: Hello!
```

Listing 1: Example of First-Class Functions

References:

- Article: Real Python: First-Class Functions
- Video: ArjanCodes: First-Class Functions
- Documentation: Python Functional Programming HOWTO

1.3 Closures and Wrapper Functions

A *closure* is a function that retains access to variables from its enclosing scope. Decorators use closures to create *wrapper functions* that add behavior before or after the original function.

```
def outer_function(msg):
    def inner_function():
        print(msg)
    return inner_function

```

```
6 closure = outer_function("Hello from closure!")
7 closure()  # Outputs: Hello from closure!
```

Listing 2: Example of a Closure

In decorators, the wrapper function is the inner function that wraps the original function, allowing us to add logging, error handling, or other features.

References:

- Video: Tech With Tim: Closures in Python
- Article: Programiz: Python Closures
- Book: *Python Cookbook* by David Beazley, Recipe 7.5

1.4 Why Use Decorators in Database Operations?

Database operations involve repetitive tasks like opening connections, logging queries, handling transactions, and caching results. Decorators abstract these tasks into reusable components, reducing boilerplate code and improving maintainability. For example, instead of writing connection handling code in every function, a decorator can handle it automatically.

References:

- Article: Towards Data Science: Decorators for Data Science
- Video: PyCon: Decorators and Context Managers

2 Setting Up the Environment

2.1 Installing Dependencies

To complete this project, ensure the following are installed:

- **Python 3.8+**: Download from python.org. Verify with `python -version`.
- **SQLite**: Included in Pythons `sqlite3` module.
- **IDE**: Use PyCharm or VS Code with the Python extension.
- **Git**: Install from git-scm.com for version control.
- **Optional Tools**: Install `mypy` (`pip install mypy`) for type checking and `pylint` (`pip install pylint`) for linting.

References:

- Documentation: Python Installation Guide
- Documentation: Python sqlite3 Module
- Guide: VS Code Python Setup
- Book: Pro Git Book
- Documentation: MyPy, Pylint

2.2 Creating a Sample SQLite Database

Create a SQLite database `users.db` with a `users` table for testing.

```python
import sqlite3
from typing import List, Tuple

def create_sample_database() -> None:
    """
    Creates a SQLite database with a users table and sample data.
    """
    try:
        conn = sqlite3.connect('users.db')
        cursor = conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                email TEXT NOT NULL UNIQUE
            )
        ''')
        sample_users: List[Tuple[str, str]] = [
            ('Alice Smith', 'alice.smith@example.com'),
            ('Bob Johnson', 'bob.johnson@example.com'),
            ('Charlie Brown', 'charlie.brown@example.com')
        ]
        cursor.executemany('INSERT OR IGNORE INTO users (name, email)
            VALUES (?, ?)', sample_users)
        conn.commit()
        print("Sample database created successfully.")
    except sqlite3.Error as e:
        print(f"Error creating database: {e}")
        raise
    finally:
        conn.close()

if __name__ == "__main__":
    create_sample_database()
```

Listing 3: setup_database.py

Run with `python setup_database.py`.

References:

- Tutorial: SQLite Python Tutorial
- Documentation: SQLite CREATE TABLE

2.3 Project Structure and Version Control

Create a GitHub repository `alx-backend-python` with a directory `python-decorators-0x01`. Initialize Git:

```
git init
git add .
git commit -m "Initialize Python decorators project"
git remote add origin <your-repo-url>
git push -u origin main
```

References:

- Guide: GitHub: Creating a Repository

3 Core Concepts

3.1 Python Functions and Decorators

Functions are first-class objects, enabling decorators. The `functools.wraps` function preserves function metadata.

References:

- Documentation: Python Functions
- Documentation: Python functools Module

3.2 SQLite and Database Operations

SQLite is a serverless database. Connections manage database access, and cursors execute queries.

References:

- Documentation: SQLite Documentation
- Tutorial: SQLite Python Tutorial

3.3 Logging and Monitoring

Logging tracks program execution for debugging and auditing. Pythons `logging` module supports file and console output.

References:

- Video: Python Logging Tutorial
- Documentation: Python logging Module

3.4 Transactions and Error Handling

Transactions ensure data consistency by grouping operations. Error handling uses `try-except` to manage failures.

References:

- Documentation: SQLite Transactions
- Tutorial: Python Exceptions

3.5 Caching and Performance Optimization

Caching stores results to avoid redundant computations. We use a dictionary for simplicity, but `functools.lru_cache` is an alternative.

References:

- Article: Real Python: Caching in Python
- Documentation: functools.lru_cache

3.6 Retry Mechanisms

Retries handle transient errors (e.g., database locks) by attempting operations multiple times.

References:

- Article: Sicara: Python Retry Decorators
- Library: Tenacity Retry Library

4 Task Implementations

4.1 Task 0: Logging Database Queries

4.1.1 Objective

Create a `log_queries` decorator to log SQL queries with timestamps and parameters.

4.1.2 Concepts

- **Logging**: Tracks queries for debugging.
- **Decorator Flexibility**: Handles various function signatures with `*args` and `**kwargs`.
- **Thread Safety**: Ensures logging is safe in concurrent applications.

References:

- Video: Python Logging Tutorial
- Article: Real Python: Logging
- Documentation: Logging Thread Safety

4.1.3 Implementation

1. Configure a thread-safe logger with file and console handlers.
2. Define `log_queries` to log the query and parameters.
3. Use type hints and `functools.wraps` for metadata preservation.
4. Log execution time for performance monitoring.

```
import sqlite3
import functools
import logging
from typing import Callable, Any, Tuple, List
from datetime import datetime

# Configure thread-safe logging
logger = logging.getLogger('db_queries')
logger.setLevel(logging.INFO)
file_handler = logging.FileHandler('database_queries.log')
file_handler.setFormatter(logging.Formatter(
    '%(asctime)s - %(levelname)s - %(message)s'
))
logger.addHandler(file_handler)
console_handler = logging.StreamHandler()
console_handler.setFormatter(logging.Formatter(
    '%(asctime)s - %(levelname)s - %(message)s'
))
```

```
logger.addHandler(console_handler)

def log_queries(func: Callable[..., Any]) -> Callable[..., Any]:
    """
    Decorator to log SQL queries with parameters and execution time.

    Args:
        func: Function to decorate.
    Returns:
        Callable: Wrapped function with logging.
    """
    @functools.wraps(func)
    def wrapper(*args, **kwargs) -> Any:
        query = args[0] if args else "Unknown query"
        params = args[1:] if len(args) > 1 else kwargs.get('params', ())
        start_time = datetime.now()

        logger.info(f"Executing query: {query} with params: {params}")
        try:
            result = func(*args, **kwargs)
            execution_time = (datetime.now() - start_time).total_seconds()
            logger.info(f"Query completed in {execution_time:.3f} seconds")
            return result
        except Exception as e:
            logger.error(f"Query failed: {e}")
            raise
    return wrapper

@log_queries
def fetch_all_users(query: str, params: Tuple = ()) -> List[Tuple]:
    """
    Fetch users from the database.

    Args:
        query: SQL query string.
        params: Query parameters.
    Returns:
        List of user records.
    """
    conn = sqlite3.connect('users.db')
    try:
        cursor = conn.cursor()
        cursor.execute(query, params)
        return cursor.fetchall()
    finally:
        conn.close()

if __name__ == "__main__":
    try:
        users = fetch_all_users("SELECT * FROM users")
        for user in users:
            print(user)
    except sqlite3.Error as e:
        print(f"Database error: {e}")
```

Listing 4: 0-log_queries.py

4.1.4 Testing

1. Run `setup_database.py` to create `users.db`.
2. Execute `python 0-log_queries.py`.
3. Expected output:

```
2025-05-20 12:57:00,123 - INFO - Executing query: SELECT * FROM users with params: ()
2025-05-20 12:57:00,125 - INFO - Query completed in 0.002 seconds
(1, 'Alice Smith', 'alice.smith@example.com')
(2, 'Bob Johnson', 'bob.johnson@example.com')
(3, 'Charlie Brown', 'charlie.brown@example.com')
```

4. Verify `database_queries.log` for logs.

4.1.5 Extra Example: Logging with User Context

Add user context to logs:

```python
def log_with_context(user_id: int) -> Callable:
    def decorator(func: Callable[..., Any]) -> Callable[..., Any]:
        @functools.wraps(func)
        def wrapper(*args, **kwargs) -> Any:
            query = args[0] if args else "Unknown query"
            logger.info(f"User {user_id} executing query: {query}")
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

Listing 5: Logging with User Context

4.2 Task 1: Database Connection Management

4.2.1 Objective

Create a `with_db_connection` decorator to manage SQLite connections.

4.2.2 Concepts

- **Connection Management**: Ensures connections are opened and closed properly.
- **Context Managers**: Mimic `with` statement behavior.
- **Type Hints**: Enhance code clarity.

References:

- Tutorial: SQLite Connection Management
- Documentation: Python Context Managers
- Documentation: Python Typing

4.2.3 Implementation

1. Define `with_db_connection` with type hints.
2. Use a context manager for connection handling.
3. Log connection events.
4. Ensure thread safety with thread-local connections.

```python
import sqlite3
import functools
import logging
from typing import Callable, Any, Optional
from contextlib import contextmanager

# Configure logging
logger = logging.getLogger('db_connection')
logger.setLevel(logging.INFO)
file_handler = logging.FileHandler('connection.log')
file_handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)
    s - %(message)s'))
logger.addHandler(file_handler)
logger.addHandler(logging.StreamHandler())

@contextmanager
def db_connection(db_name: str) -> sqlite3.Connection:
    """
    Context manager for SQLite connections.

    Args:
        db_name: Database file path.
    Yields:
        sqlite3.Connection: Database connection.
    """
    conn = None
    try:
        conn = sqlite3.connect(db_name, check_same_thread=True)
        logger.info(f"Opened connection to {db_name}")
        yield conn
    except sqlite3.Error as e:
        logger.error(f"Connection error: {e}")
        raise
    finally:
        if conn:
            conn.close()
            logger.info(f"Closed connection to {db_name}")

def with_db_connection(func: Callable[..., Any]) -> Callable[..., Any]:
    """
    Decorator to manage SQLite connections.

    Args:
        func: Function to decorate.
    Returns:
        Callable: Wrapped function with connection management.
    """
    @functools.wraps(func)
    def wrapper(*args, **kwargs) -> Any:
        with db_connection('users.db') as conn:
```

```
            return func(conn, *args, **kwargs)
    return wrapper

@with_db_connection
def get_user_by_id(conn: sqlite3.Connection, user_id: int) -> Optional[
    Tuple]:
    """
    Fetch a user by ID.

    Args:
        conn: SQLite connection.
        user_id: User ID.
    Returns:
        User record or None.
    """
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
    return cursor.fetchone()

if __name__ == "__main__":
    try:
        user = get_user_by_id(user_id=1)
        print(user)
    except sqlite3.Error as e:
        print(f"Database error: {e}")
```

Listing 6: 1-with_db_connection.py

4.2.4 Testing

1. Run `setup_database.py`.
2. Execute `python 1-with_db_connection.py`.
3. Expected output:

```
2025-05-20 12:57:00,123 - INFO - Opened connection to users.db
(1, 'Alice Smith', 'alice.smith@example.com')
2025-05-20 12:57:00,125 - INFO - Closed connection to users.db
```

4.2.5 Extra Example: Connection Pooling

Use a connection pool for high-concurrency applications:

```
from sqlite3 import dbapi2 as sqlite3

def with_pooled_connection(func: Callable[..., Any]) -> Callable[...,
    Any]:
    @functools.wraps(func)
    def wrapper(*args, **kwargs) -> Any:
        with sqlite3.connect('users.db', uri=True) as conn:
            return func(conn, *args, **kwargs)
    return wrapper
```

Listing 7: Connection Pooling

4.3 Task 2: Transaction Management

4.3.1 Objective

Create a `transactional` decorator to manage database transactions, committing on success or rolling back on failure.

4.3.2 Concepts

- **Transactions**: Ensure data consistency.
- **Decorator Stacking**: Combine with `with_db_connection`.

References:

- Documentation: SQLite Transactions
- Article: Real Python: Transactions

4.3.3 Implementation

1. Reuse `with_db_connection`.
2. Define `transactional` to commit or rollback.
3. Use type hints and logging.

```python
import sqlite3
import functools
import logging
from typing import Callable, Any, Optional
from contextlib import contextmanager

# Configure logging
logger = logging.getLogger('db_transaction')
logger.setLevel(logging.INFO)
file_handler = logging.FileHandler('transaction.log')
file_handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)s - %(message)s'))
logger.addHandler(file_handler)
logger.addHandler(logging.StreamHandler())

@contextmanager
def db_connection(db_name: str) -> sqlite3.Connection:
    conn = None
    try:
        conn = sqlite3.connect(db_name, check_same_thread=True)
        logger.info(f"Opened connection to {db_name}")
        yield conn
    except sqlite3.Error as e:
        logger.error(f"Connection error: {e}")
        raise
    finally:
        if conn:
            conn.close()
            logger.info(f"Closed connection to {db_name}")

def with_db_connection(func: Callable[..., Any]) -> Callable[..., Any]:
    @functools.wraps(func)
    def wrapper(*args, **kwargs) -> Any:
```

```python
        with db_connection('users.db') as conn:
            return func(conn, *args, **kwargs)
    return wrapper

def transactional(func: Callable[..., Any]) -> Callable[..., Any]:
    """
    Decorator to manage database transactions.

    Args:
        func: Function to decorate.
    Returns:
        Callable: Wrapped function with transaction management.
    """
    @functools.wraps(func)
    def wrapper(conn: sqlite3.Connection, *args, **kwargs) -> Any:
        try:
            result = func(conn, *args, **kwargs)
            conn.commit()
            logger.info("Transaction committed")
            return result
        except Exception as e:
            conn.rollback()
            logger.error(f"Transaction rolled back: {e}")
            raise
    return wrapper

@with_db_connection
@transactional
def update_user_email(conn: sqlite3.Connection, user_id: int, new_email
    : str) -> int:
    """
    Update a user's email.

    Args:
        conn: SQLite connection.
        user_id: User ID.
        new_email: New email address.
    Returns:
        Number of affected rows.
    """
    cursor = conn.cursor()
    cursor.execute("UPDATE users SET email = ? WHERE id = ?", (
       new_email, user_id))
    return cursor.rowcount

if __name__ == "__main__":
    try:
        updated_rows = update_user_email(user_id=1, new_email='
           crawford@example.com')
        print(f"Updated {updated_rows} row(s)")
    except sqlite3.Error as e:
        print(f"Database error: {e}")
```

Listing 8: 2-transactional.py

4.3.4 Testing

1. Run `setup_database.py`.

2. Execute `python 2-transactional.py`.
3. Expected output:

```
2025-05-20 12:57:00,123 - INFO - Opened connection to users.db
2025-05-20 12:57:00,125 - INFO - Transaction committed
Updated 1 row(s)
2025-05-20 12:57:00,126 - INFO - Closed connection to users.db
```

4.3.5 Extra Example: Multi-Statement Transaction

Handle multiple updates in one transaction:

```python
@with_db_connection
@transactional
def update_user_details(conn: sqlite3.Connection, user_id: int, name: str, email: str) -> int:
    cursor = conn.cursor()
    cursor.execute("UPDATE users SET name = ?, email = ? WHERE id = ?", (name, email, user_id))
    return cursor.rowcount
```

Listing 9: Multi-Statement Transaction

4.4 Task 3: Retry Database Queries

4.4.1 Objective

Create a `retry_on_failure` decorator to retry operations on transient errors.

4.4.2 Concepts

- **Transient Errors**: Temporary issues like database locks.
- **Parameterized Decorators**: Accept arguments like retries and delay.

References:

- Article: Sicara: Retry Decorators
- Library: Tenacity

4.4.3 Implementation

1. Define a parameterized decorator with `retries` and `delay`.
2. Implement retry logic with exponential backoff.
3. Log retry attempts.

```python
import sqlite3
import functools
import logging
import time
from typing import Callable, Any, List, Tuple
from contextlib import contextmanager

# Configure logging
logger = logging.getLogger('db_retry')
```

```
logger.setLevel(logging.INFO)
file_handler = logging.FileHandler('retry.log')
file_handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)s - %(message)s'))
logger.addHandler(file_handler)
logger.addHandler(logging.StreamHandler())

@contextmanager
def db_connection(db_name: str) -> sqlite3.Connection:
    conn = None
    try:
        conn = sqlite3.connect(db_name, check_same_thread=True)
        logger.info(f"Opened connection to {db_name}")
        yield conn
    except sqlite3.Error as e:
        logger.error(f"Connection error: {e}")
        raise
    finally:
        if conn:
            conn.close()
            logger.info(f"Closed connection to {db_name}")

def with_db_connection(func: Callable[..., Any]) -> Callable[..., Any]:
    @functools.wraps(func)
    def wrapper(*args, **kwargs) -> Any:
        with db_connection('users.db') as conn:
            return func(conn, *args, **kwargs)
    return wrapper

def retry_on_failure(retries: int = 3, delay: float = 1,RULE
    """
    Decorator to retry database operations on transient errors.

    Args:
        retries: Number of retry attempts.
        delay: Delay between retries in seconds.
    Returns:
        Callable: Wrapped function with retry logic.
    """
    def decorator(func: Callable[..., Any]) -> Callable[..., Any]:
        @functools.wraps(func)
        def wrapper(*args, **kwargs) -> Any:
            last_exception = None
            for attempt in range(retries + 1):
                try:
                    return func(*args, **kwargs)
                except sqlite3.OperationalError as e:
                    last_exception = e
                    if attempt < retries:
                        logger.info(f"Retry {attempt + 1}/{retries} after {delay}s: {e}")
                        time.sleep(delay)
            raise last_exception
        return wrapper
    return decorator

@with_db_connection
@retry_on_failure(retries=3, delay=1)
```

```python
def fetch_users_with_retry(conn: sqlite3.Connection) -> List[Tuple]:
    """
    Fetch users with retry on failure.

    Args:
        conn: SQLite connection.
    Returns:
        List of user records.
    """
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    return cursor.fetchall()

if __name__ == "__main__":
    try:
        users = fetch_users_with_retry()
        for user in users:
            print(user)
    except sqlite3.Error as e:
        print(f"Failed after retries: {e}")
```

Listing 10: 3-retry_on_failure.py

4.4.4 Testing

1. Run `setup_database.py`.
2. Execute `python 3-retry_on_failure.py`.
3. Expected output:

```
(1, 'Alice Smith', 'alice.smith@example.com')
(2, 'Bob Johnson', 'bob.johnson@example.com')
(3, 'Charlie Brown', 'charlie.brown@example.com')
2025-05-20 12:57:00,126 - INFO - Closed connection to users.db
```

4.4.5 Extra Example: Exponential Backoff

Use exponential backoff for retries:

```python
def retry_with_backoff(retries: int = 3, initial_delay: float = 1,
    backoff_factor: float = 2) -> Callable:
    def decorator(func: Callable[..., Any]) -> Callable[..., Any]:
        @functools.wraps(func)
        def wrapper(*args, **kwargs) -> Any:
            delay = initial_delay
            last_exception = None
            for attempt in range(retries + 1):
                try:
                    return func(*args, **kwargs)
                except sqlite3.OperationalError as e:
                    last_exception = e
                    if attempt < retries:
                        logger.info(f"Retry {attempt + 1}/{retries}
                            after {delay}s: {e}")
                        time.sleep(delay)
                        delay *= backoff_factor
```

```
            raise last_exception
        return wrapper
    return decorator
```

Listing 11: Exponential Backoff Retry

4.5 Task 4: Cache Database Queries

4.5.1 Objective

Create a `cache_query` decorator to cache query results.

4.5.2 Concepts

- **Caching**: Stores results to avoid redundant queries.
- **Cache Key**: Uses query and parameters for uniqueness.

References:

- Article: Real Python: Caching
- Documentation: functools.lru$_c$*ache*

4.5.3 Implementation

1. Define a global `query_cache` dictionary.
2. Create `cache_query` to check and store results.
3. Use a tuple of query and parameters as the cache key.

```
import sqlite3
import functools
import logging
import time
from typing import Callable, Any, List, Tuple, Dict
from contextlib import contextmanager

# Configure logging
logger = logging.getLogger('db_cache')
logger.setLevel(logging.INFO)
file_handler = logging.FileHandler('cache.log')
file_handler.setFormatter(logging.Formatter('%(asctime)s - %(levelname)
    s - %(message)s'))
logger.addHandler(file_handler)
logger.addHandler(logging.StreamHandler())

query_cache: Dict[tuple, tuple] = {}

@contextmanager
def db_connection(db_name: str) -> sqlite3.Connection:
    conn = None
    try:
        conn = sqlite3.connect(db_name, check_same_thread=True)
        logger.info(f"Opened connection to {db_name}")
        yield conn
    except sqlite3.Error as e:
        logger.error(f"Connection error: {e}")
        raise
```

```
    finally:
        if conn:
            conn.close()
            logger.info(f"Closed connection to {db_name}")

def with_db_connection(func: Callable[..., Any]) -> Callable[..., Any]:
    @functools.wraps(func)
    def wrapper(*args, **kwargs) -> Any:
        with db_connection('users.db') as conn:
            return func(conn, *args, **kwargs)
    return wrapper

def cache_query(func: Callable[..., Any]) -> Callable[..., Any]:
    """
    Decorator to cache query results.

    Args:
        func: Function to decorate.
    Returns:
        Callable: Wrapped function with caching.
    """
    @functools.wraps(func)
    def wrapper(conn: sqlite3.Connection, query: str, *args, **kwargs)
       -> Any:
        cache_key = (query, args, tuple(sorted(kwargs.items())))
        if cache_key in query_cache:
            logger.info(f"Cache hit for query: {query}")
            return query_cache[cache_key]
        logger.info(f"Cache miss for query: {query}")
        result = func(conn, query, *args, **kwargs)
        query_cache[cache_key] = result
        return result
    return wrapper

@with_db_connection
@cache_query
def fetch_users_with_cache(conn: sqlite3.Connection, query: str) ->
   List[Tuple]:
    """
    Fetch users with caching.

    Args:
        conn: SQLite connection.
        query: SQL query string.
    Returns:
        List of user records.
    """
    cursor = conn.cursor()
    cursor.execute(query)
    return cursor.fetchall()

if __name__ == "__main__":
    try:
        users = fetch_users_with_cache(query="SELECT * FROM users")
        print("First call:", users)
        users_again = fetch_users_with_cache(query="SELECT * FROM users
           ")
        print("Second call:", users_again)
```

```
    except sqlite3.Error as e:
        print(f"Database error: {e}")
```

Listing 12: 4-cache_query.py

4.5.4 Testing

1. Run `setup_database.py`.
2. Execute `python 4-cache_query.py`.
3. Expected output:

```
2025-05-20 12:57:00,123 - INFO - Opened connection to users.db
2025-05-20 12:57:00,124 - INFO - Cache miss for query: SELECT * FROM users
First call: [(1, 'Alice Smith', 'alice.smith@example.com'), ...]
2025-05-20 12:57:00,125 - INFO - Closed connection to users.db
2025-05-20 12:57:00,126 - INFO - Cache hit for query: SELECT * FROM users
Second call: [(1, 'Alice Smith', 'alice.smith@example.com'), ...]
```

4.5.5 Extra Example: Cache with Expiration

Add cache expiration:

```
def cache_with_expiration(timeout: float = 60) -> Callable:
    def decorator(func: Callable[..., Any]) -> Callable[..., Any]:
        @functools.wraps(func)
        def wrapper(conn: sqlite3.Connection, query: str, *args, **
            kwargs) -> Any:
            cache_key = (query, args, tuple(sorted(kwargs.items())))
            if cache_key in query_cache:
                result, timestamp = query_cache[cache_key]
                if time.time() - timestamp < timeout:
                    logger.info(f"Cache hit for query: {query}")
                    return result
                logger.info(f"Cache expired for query: {query}")
            logger.info(f"Cache miss for query: {query}")
            result = func(conn, query, *args, **kwargs)
            query_cache[cache_key] = (result, time.time())
            return result
        return wrapper
    return decorator
```

Listing 13: Cache with Expiration

5 Advanced Decorator Techniques

5.1 Class-Based Decorators

Use classes for stateful decorators:

```
class QueryLogger:
    def __init__(self, log_file: str = 'queries.log'):
        self.log_file = log_file

    def __call__(self, func: Callable[..., Any]) -> Callable[..., Any]:
```

```
        @functools.wraps(func)
        def wrapper(*args, **kwargs) -> Any:
            query = args[0] if args else "Unknown query"
            with open(self.log_file, 'a') as f:
                f.write(f"Query: {query}\n")
            return func(*args, **kwargs)
        return wrapper
```

Listing 14: Class-Based Decorator

References:

- Article: Real Python: Class Decorators

5.2 Decorator Factories

Parameterized decorators require an extra function layer:

```
def repeat(n: int) -> Callable:
    def decorator(func: Callable[..., Any]) -> Callable[..., Any]:
        @functools.wraps(func)
        def wrapper(*args, **kwargs) -> Any:
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator
```

Listing 15: Decorator Factory

References:

- Article: Real Python: Decorators with Arguments

5.3 Chaining Decorators

Stack multiple decorators:

```
@with_db_connection
@log_queries
@cache_query
def complex_query(conn: sqlite3.Connection, query: str) -> List[Tuple]:
    cursor = conn.cursor()
    cursor.execute(query)
    return cursor.fetchall()
```

Listing 16: Chained Decorators

6 Best Practices and Debugging

6.1 Best Practices

- Use `functools.wraps` to preserve metadata.
- Implement thread-safe logging and connection handling.
- Use type hints for clarity.
- Log detailed information for debugging.

6.2 Debugging

- Use `pdb` or IDE debuggers.
- Test edge cases (e.g., invalid queries, missing database).
- Profile with `time.time()` for performance.

References:

- Documentation: Python pdb
- Guide: Real Python: Debugging with pdb

7 Learning Resources

- **Books**:
  - *Fluent Python* by Luciano Ramalho
  - *Python Cookbook* by David Beazley
- **Courses**:
  - Pluralsight: Advanced Python
  - Udemy: Python Design Patterns
- **Communities**:
  - Python Discord
  - Stack Overflow Python

8 Submission Guidelines

1. Create repository `alx-backend-python`.
2. Add directory `python-decorators-0x01`.
3. Include files: `0-log_queries.py`, `1-with_db_connection.py`, `2-transactional.py`, `3-retry_on_failu`
   `4-cache_query.py`.
4. Commit and push:

```
git add .
git commit -m "Completed Python decorators project"
git push origin main
```

5. Request manual QA review by May 26, 2025.