Python

Python Context Managers

Elegant Resource Management with the «with» Statement

March 17, 2025

Source Code

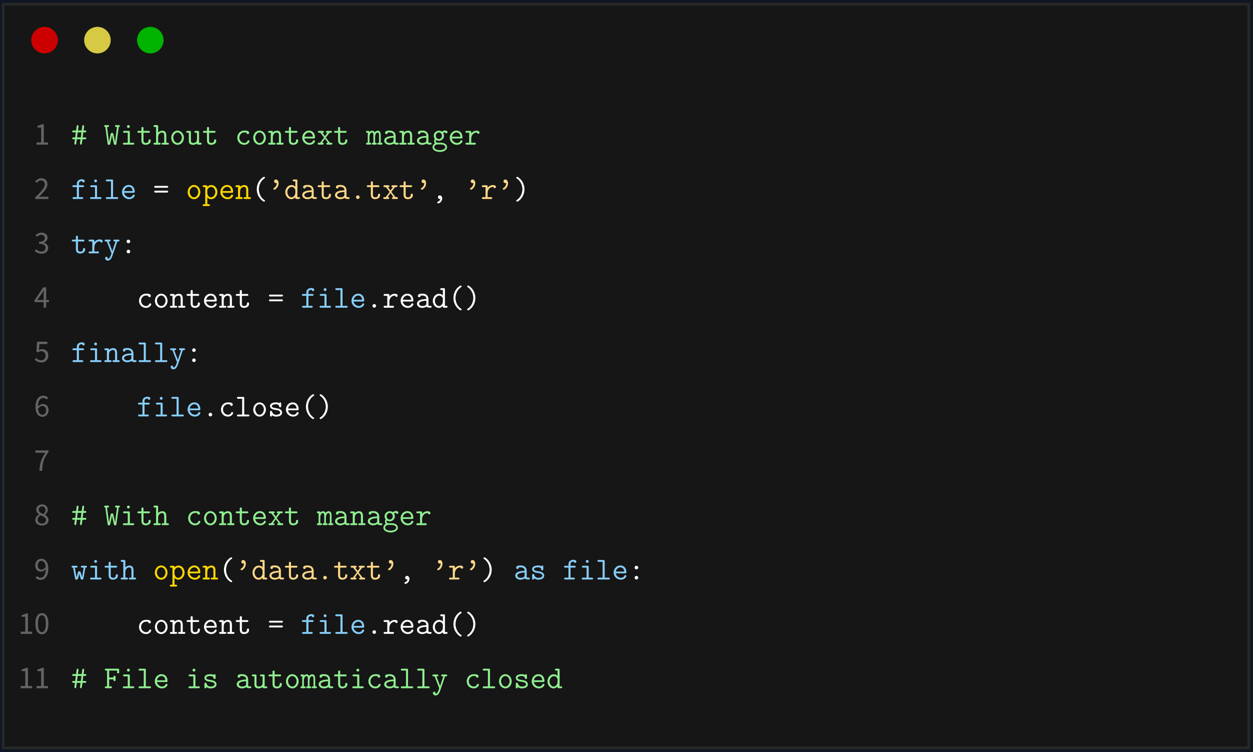
1. What Are Context Managers?

Context managers in Python provide an elegant way to manage resources automatically. Using the **with** statement, they handle setup and cleanup operations, ensuring resources are properly acquired and released, even when errors occur.

- **Resource Safety:** Automatic cleanup even during exceptions
- **Code Simplicity:** Eliminates boilerplate try/finally blocks
- **Error Handling:** Built-in mechanism for exception management

1.1. The «with» Statement

Compare traditional resource management with context managers:



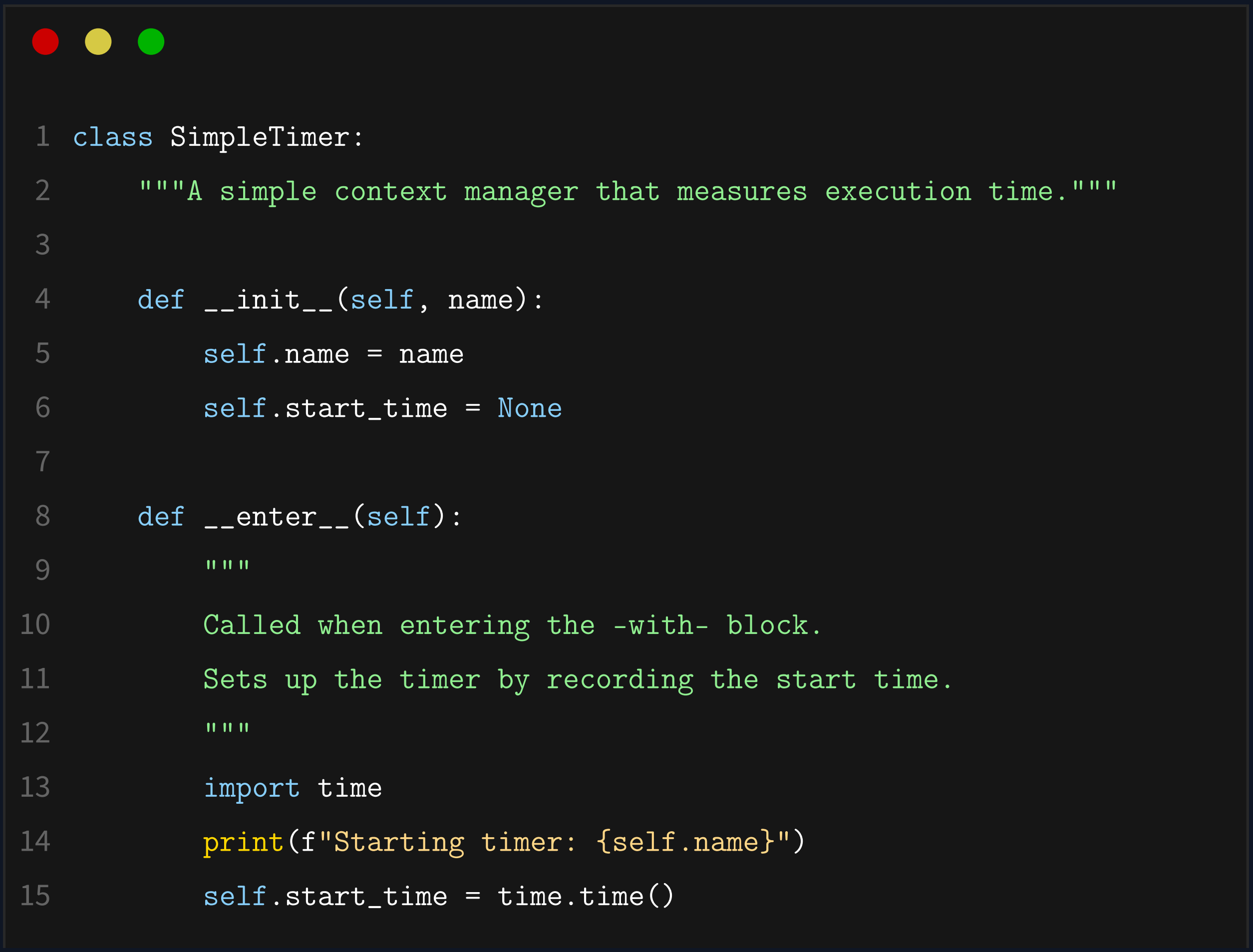
```python
# Without context manager
file = open(’data.txt’, ’r’)
try:
    content = file.read()
finally:
    file.close()

# With context manager
with open(’data.txt’, ’r’) as file:
    content = file.read()
# File is automatically closed
```

2. Creating Context Managers

2.1. Class-Based Implementation

To implement a context manager as a class, you need to define the **__enter__** and **__exit__** methods:

- **__enter__**: Called when entering the **with** block - handles setup
- **__exit__**: Called when exiting the **with** block - handles cleanup



```python
class SimpleTimer:
    """A simple context manager that measures execution time."""

    def __init__(self, name):
        self.name = name
        self.start_time = None

    def __enter__(self):
        """
        Called when entering the -with- block.
        Sets up the timer by recording the start time.
        """
        import time
        print(f"Starting timer: {self.name}")
        self.start_time = time.time()
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
        return self  # This object will be assigned to the variable after -as-

    def __exit__(self, exc_type, exc_value, traceback):
        """
        Called when exiting the -with- block (whether normally or due to exception).
        Calculates and displays the elapsed time.

        Parameters:
        - exc_type: Type of exception that occurred (or None if no exception)
        - exc_value: The exception object (or None)
        - traceback: The traceback object (or None)
        """
        import time
        elapsed = time.time() - self.start_time

        if exc_type is None:
            # No exception occurred
            print(f"Timer {self.name} finished: {elapsed:.4f} seconds")
        else:
            # An exception occurred
            print(f"Timer {self.name} aborted: {elapsed:.4f} seconds")
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
🌐 www.asanchezyali.com

```
            print(f"Exception: {exc_type.__name__}: {exc_value}")

        # Return False to let exceptions propagate
        # Return True would suppress the exception
        return False

# Using our custom context manager
with SimpleTimer("calculation") as timer:
    print("Performing calculation...")
    # Simulate some work
    total = sum(i**2 for i in range(1000000))
    print(f"Result: {total}")

# The __exit__ method is called automatically when leaving the with block
print("After the with block")
```

2.2. Function-Based Implementation

Using the **@contextmanager** decorator for a simpler approach:

```
from contextlib import contextmanager

@contextmanager
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

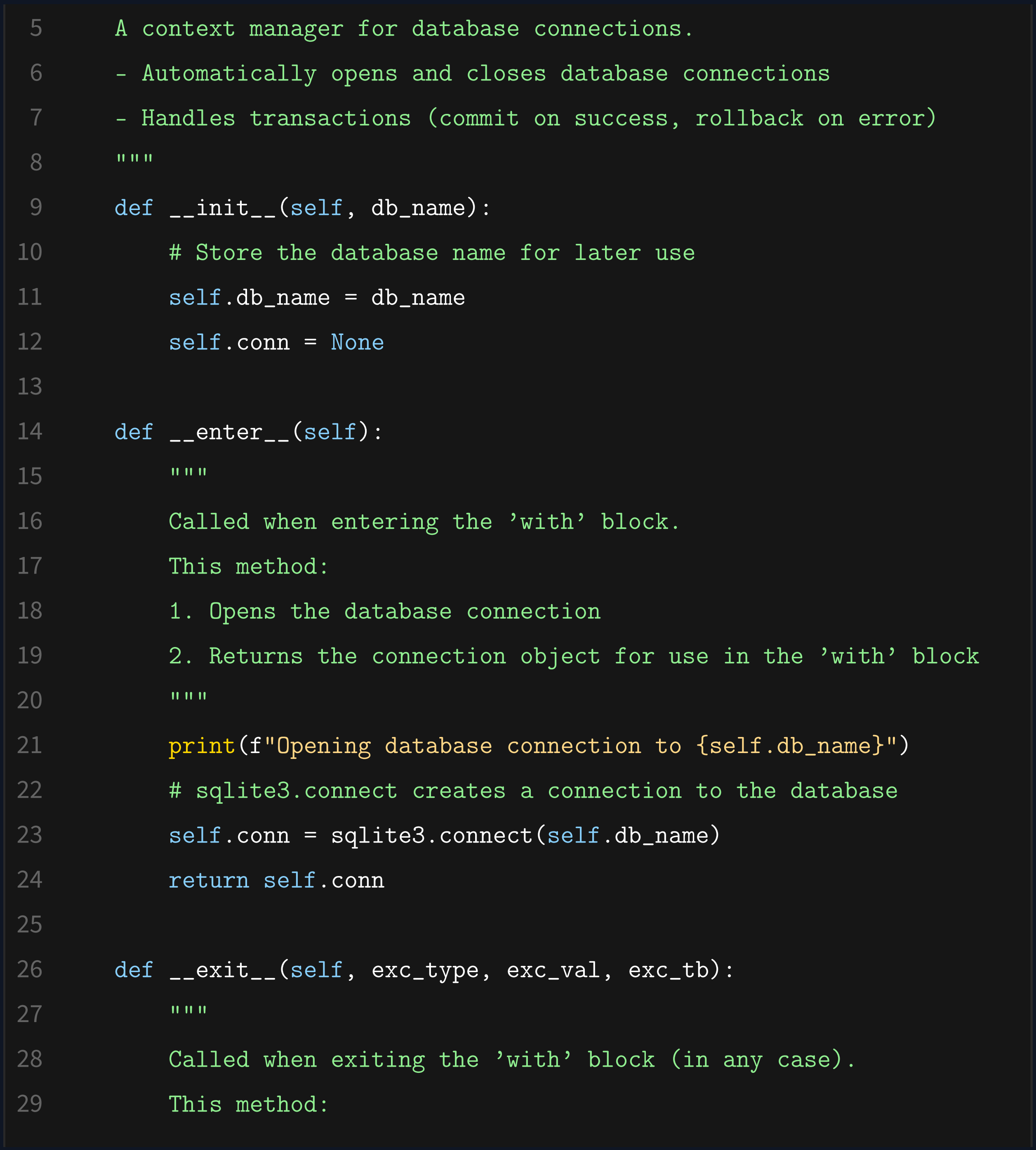
```python
def file_manager(filename, mode):
    try:
        # Setup code
        f = open(filename, mode)
        yield f  # Yield the resource
    finally:
        # Cleanup code
        f.close()

# Usage
with file_manager(’example.txt’, ’r’) as f:
    content = f.read()
```

3. Practical Applications

3.1. Database Transactions

A database context manager handles connections and transactions safely. This example explains how it works without requiring database expertise:

```python
import sqlite3

class DatabaseConnection:
    """
```

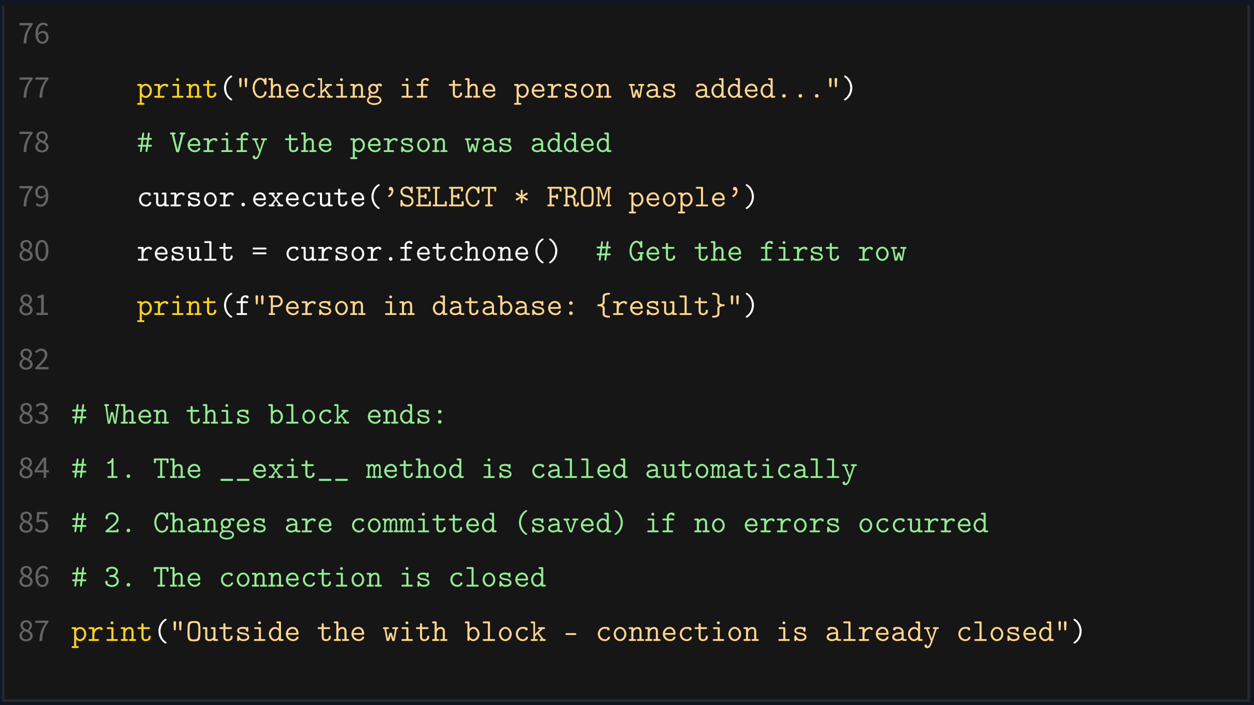


```python
    A context manager for database connections.
    - Automatically opens and closes database connections
    - Handles transactions (commit on success, rollback on error)
    """
    def __init__(self, db_name):
        # Store the database name for later use
        self.db_name = db_name
        self.conn = None

    def __enter__(self):
        """
        Called when entering the ’with’ block.
        This method:
        1. Opens the database connection
        2. Returns the connection object for use in the ’with’ block
        """
        print(f"Opening database connection to {self.db_name}")
        # sqlite3.connect creates a connection to the database
        self.conn = sqlite3.connect(self.db_name)
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        """
        Called when exiting the ’with’ block (in any case).
        This method:
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
        1. Checks if there was an exception (error)
        2. Rolls back changes if there was an error
        3. Commits changes if everything was successful
        4. Always closes the connection

        Parameters:
        - exc_type: The type of exception that occurred (or None if no exception)
        - exc_val: The exception object (or None)
        - exc_tb: The traceback object (or None)
        """
        if self.conn:
            if exc_type:  # If an exception/error occurred in the ’with’ block
                print("Error occurred! Rolling back changes...")
                # rollback() undoes all changes made in the current transaction
                self.conn.rollback()
            else:  # If everything went well (no exceptions)
                print("Operations successful! Saving changes...")
                # commit() saves all changes made in the current transaction
                self.conn.commit()

            print("Closing database connection")
```

```
            # Always close the connection to free up resources
            self.conn.close()

        # Return False to let exceptions propagate (be raised)
        # Return True would suppress the exception
        return False

# Simple usage example with an in-memory database (no file needed)
with DatabaseConnection(’:memory:’) as conn:
    # Create a cursor object to execute SQL commands
    cursor = conn.cursor()

    print("Creating a table...")
    # Create a simple table to store people
    cursor.execute(’’’
    CREATE TABLE people (
        id INTEGER PRIMARY KEY,
        name TEXT,
        age INTEGER
    )’’’)

    print("Adding a person to the database...")
    # Insert a person into the table
    cursor.execute(’INSERT INTO people (name, age) VALUES (?, ?)’,
                  (’John’, 25))
```

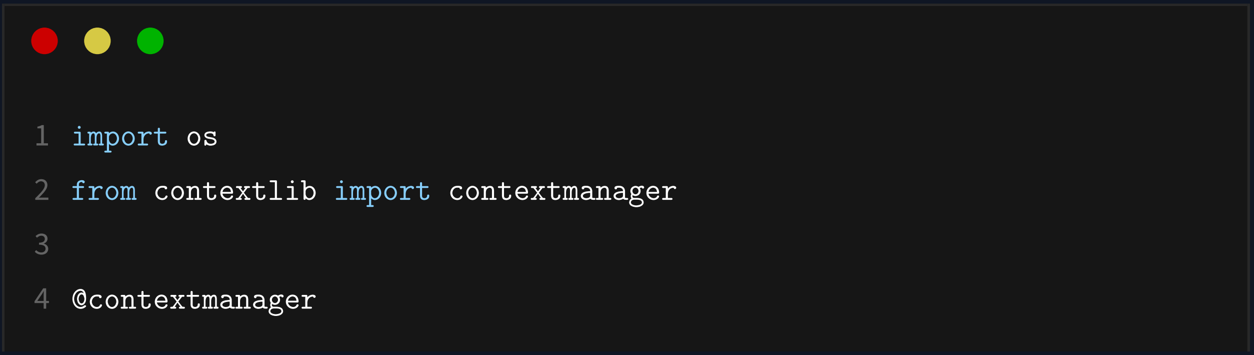


```python

    print("Checking if the person was added...")
    # Verify the person was added
    cursor.execute('SELECT * FROM people')
    result = cursor.fetchone()  # Get the first row
    print(f"Person in database: {result}")

# When this block ends:
# 1. The __exit__ method is called automatically
# 2. Changes are committed (saved) if no errors occurred
# 3. The connection is closed
print("Outside the with block - connection is already closed")
```

This example shows how context managers eliminate the need to manually open and close database connections, and handle transaction management (commit/rollback) automatically.

3.2. Temporary State Changes

Temporarily modify settings or environments:



```python
import os
from contextlib import contextmanager

@contextmanager
```

```
def change_directory(path):
    """Temporarily change the working directory."""
    old_dir = os.getcwd()
    try:
        print(f"Changing directory to: {path}")
        os.chdir(path)
        yield  # Give control back to the with block
    finally:
        print(f"Changing back to: {old_dir}")
        os.chdir(old_dir)  # Always restore the original directory

# Code runs with /tmp as working directory
with change_directory("/tmp"):
    print(f"Current directory: {os.getcwd()}")
    # Do some work in the temporary directory

# Original directory is restored automatically
print("Back to the original directory")
print(f"Current directory: {os.getcwd()}")
```

4. Advanced Features

4.1. Exception Handling

Context managers can control whether exceptions are propagated or suppressed:

```python
class ExceptionHandler:
    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is not None:
            print(f"Handled: {exc_type.__name__}: {exc_val}")
            return True  # Suppress the exception
        return False  # Propagate any exception

# Exception will be suppressed
with ExceptionHandler():
    result = 1 / 0
    print("This won’t execute")
print("But this will")  # Execution continues
```

4.2. Useful Built-in Context Managers

Python provides several ready-to-use context managers:

```python
from contextlib import suppress, redirect_stdout
import io
```

```python

# Suppress specific exceptions
with suppress(FileNotFoundError):
    # No exception if file doesn’t exist
    open(’non_existent.txt’).read()

# Redirect output
f = io.StringIO()
with redirect_stdout(f):
    print("Hello, world!")
output = f.getvalue()  # Contains "Hello, world!"
print(output)
```

5. Performance Measurement

Create a simple timer context manager:

```python
import time
from contextlib import contextmanager

@contextmanager
def timer(label):
    """
```

```python
    A context manager that measures and prints the execution time
    of the code inside the ’with’ block.

    Parameters:
    - label: A descriptive name for the operation being timed
    """
    # 1. Setup phase: Record the start time
    start = time.time()
    try:
        # 2. Yield control to the with-block
        #    Note: We don’t yield a value here since we don’t need
        #    to expose any object to the with-block
        yield
    finally:
        # 3. Cleanup phase: Calculate and display elapsed time
        #    This runs even if an exception occurs in the with-block
        end = time.time()
        elapsed = end - start
        print(f"{label}: {elapsed:.4f} seconds")

# Example 1: Basic usage
with timer("Processing data"):
    # Time-consuming operation
    time.sleep(0.5)  # Simulate work with a delay

```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```python
# Example 2: Nested timers for profiling different parts of code
with timer("Complete operation"):
    # First subtask
    with timer("Data loading"):
        time.sleep(0.2)  # Simulate loading data

    # Second subtask
    with timer("Processing"):
        time.sleep(0.3)  # Simulate processing

    # Third subtask
    with timer("Saving results"):
        time.sleep(0.1)  # Simulate saving

# Output will show:
# Processing data: 0.5002 seconds
#   Data loading: 0.2001 seconds
#   Processing: 0.3002 seconds
#   Saving results: 0.1002 seconds
# Complete operation: 0.6009 seconds
```

6. Best Practices

- **Use for Resource Management:** Files, connections, locks
- **Keep Context Managers Focused:** Single responsibility

- **Handle Exceptions Properly:** Decide when to propagate or suppress
- **Document Behavior:** Especially exception handling
- **Test Both Paths:** Success and exception scenarios

7. Conclusion

Context managers make Python code cleaner, safer, and more maintainable by automating resource management and error handling. Whether using the built-in managers or creating custom ones, they provide an elegant solution for handling resources properly in all circumstances.

- **Automation:** Eliminates manual resource management
- **Safety:** Ensures resources are properly released
- **Flexibility:** Implement via classes or decorators

8. References

- Python Documentation. *The with statement*. Link
- Real Python. *Context Managers and Python's with Statement*. Link
- GeeksforGeeks. *Context Manager in Python*. Link
- DataCamp. *Writing Custom Context Managers in Python*. Link
- Python Documentation. *contextlib — Utilities for with-statement contexts*. Link
- Python Morsels. *Creating a context manager in Python*. Link

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- Python Tutorial. *Python Context Manager*. Link
- Code Rabbit. *Guide To Python Context Managers: Beyond Files*. Link
- Every Day Codes *Python Context Managers: Beyond in Depth*. Link
- Translated, Edited and written in collaboration with AI.

Ready to Level Up Your Python?

You’ve learned something powerful today.

Try creating a context manager in your next project

and watch your code become cleaner and safer!