Advanced Python Programming

LESSON 6: Context Managers & Basic Resource Handling

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand the context manager protocol and its purpose in resource management.
- Create custom context managers using **enter** and **exit** methods.
- Use the @contextmanager decorator from contextlib for simpler implementations.
- Apply context managers for safe file operations, configuration management, and basic resource cleanup.

Lesson Outline:

I. Understanding Context Managers (15 min)

Mental Model: Context Managers as Responsible Assistants

Think of a context manager like a responsible personal assistant:

- **Setup**: They prepare everything you need before you start working
- **Protection**: They handle problems while you work
- **Cleanup**: They clean up afterwards, even if something goes wrong
- **Reliability**: They ALWAYS clean up, no matter what happens

Context managers provide a clean way to manage resources by ensuring proper setup and teardown, even when exceptions occur.

Step 1: The problem context managers solve

```
# Problem: Manual resource management is error-prone
def risky_file_operation():
    """What happens without context managers."""
    file = None
    try:
        print("Opening file...")
        file = open("example.txt", "w")

        print("Writing to file...")
        file.write("Hello, World!")

        # What if an exception happens here?
        # raise ValueError("Something went wrong!")

        print("Closing file...")
        file.close()  # This might not get called!

    except Exception as e:
        print(f"Error occurred: {e}")
```

```
        # File might still be open!
        if file:
            file.close()  # We have to remember to do this

# The solution: Context managers guarantee cleanup
def safe_file_operation():
    """Using built-in context manager."""
    try:
        print("Using context manager...")
        with open("example.txt", "w") as file:
            print("Writing to file...")
            file.write("Hello, World!")

            # Even if exception happens here...
            # raise ValueError("Something went wrong!")

            # File is AUTOMATICALLY closed
        print("File automatically closed!")

    except Exception as e:
        print(f"Error occurred: {e}")
        print("But file was still closed properly!")

# Demonstrate both approaches
print("=== Manual Resource Management ===")
risky_file_operation()

print("\n=== Context Manager ===")
safe_file_operation()
```

Key Insight: Context managers guarantee that cleanup happens, even when things go wrong.

The Context Manager Protocol:

- **enter(self)**: Called when entering the `with` block, returns the resource
- **exit(self, exc_type, exc_value, traceback)**: Called when exiting, handles cleanup
- **Exception handling**: **exit** receives exception info and can suppress exceptions

Step 2: Understanding the protocol

```
class SimpleContextManager:
    """A simple context manager to understand the protocol."""

    def __init__(self, name):
        self.name = name
        print(f"Created context manager: {self.name}")

    def __enter__(self):
        print(f"Entering context: {self.name}")
        print("Setting up resources...")
        return f"resource_for_{self.name}"  # This is what 'as' captures
```

```
    def __exit__(self, exc_type, exc_value, traceback):
        print(f"Exiting context: {self.name}")

        if exc_type is not None:
            print(f"An exception occurred: {exc_type.__name__}:
{exc_value}")
            print("Cleaning up after exception...")
        else:
            print("Normal cleanup...")

        print("Resources cleaned up!")
        # Return False to let exceptions propagate
        return False

# Usage - understanding the flow
print("=== Context Manager Flow ===")
try:
    with SimpleContextManager("demo") as resource:
        print(f"Inside context, using: {resource}")
        print("Doing some work...")
        # Uncomment to see exception handling
        # raise ValueError("Oops!")
        print("Work completed!")
    print("After context manager")

except Exception as e:
    print(f"Caught exception: {e}")

print("Program continues...")
```

Understanding the Parameters in exit:

- `exc_type`: The exception class (or None if no exception)
- `exc_value`: The exception instance (or None)
- `traceback`: The traceback object (or None)

Basic context manager example:

```
class FileManager:
    """Custom file context manager with detailed logging."""

    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None
        print(f"FileManager created for: {filename}")

    def __enter__(self):
        print(f"Opening file: {self.filename} in mode '{self.mode}'")
        try:
```

```
            self.file = open(self.filename, self.mode)
            print(f" File opened successfully")
            return self.file
        except Exception as e:
            print(f" Failed to open file: {e}")
            raise

    def __exit__(self, exc_type, exc_value, traceback):
        print(f"Closing file: {self.filename}")

        if self.file:
            try:
                self.file.close()
                print(f" File closed successfully")
            except Exception as e:
                print(f" Error closing file: {e}")

        # Handle exceptions that occurred in the with block
        if exc_type is not None:
            print(f"Exception occurred in with block:")
            print(f"  Type: {exc_type.__name__}")
            print(f"  Message: {exc_value}")
            print(f"  File was still closed properly!")
            # Return False to propagate the exception
            return False

        return True

# Usage with success
print("=== Successful File Operation ===")
with FileManager("test.txt", "w") as f:
    f.write("Hello, World!")
    f.write("\nThis is a test file.")
    print("File operations completed")

# Usage with exception
print("\n=== File Operation with Exception ===")
try:
    with FileManager("test.txt", "r") as f:
        content = f.read()
        print(f"Read: {content}")
        raise ValueError("Something went wrong while processing!")

except ValueError as e:
    print(f"Caught exception: {e}")

print("Program continues normally")
```

Commentary:

Context managers provide guaranteed resource cleanup and make code more readable by clearly separating setup, usage, and cleanup phases.

II. The contextlib Module (20 min)

Mental Model: @contextmanager as a Smart Wrapper

Think of `@contextmanager` like a smart wrapper that converts any function into a responsible assistant:

- **Before yield**: Setup phase (like **enter**)
- **yield**: "Hand over control" - what gets returned to the with block
- **After yield**: Cleanup phase (like **exit**, runs even with exceptions)

The contextlib module provides utilities for creating and working with context managers more easily.

Step 1: Understanding @contextmanager

```
from contextlib import contextmanager

# Traditional class-based approach (verbose)
class TimerClass:
    def __init__(self, name):
        self.name = name
        self.start_time = None

    def __enter__(self):
        import time
        print(f"Starting timer: {self.name}")
        self.start_time = time.time()
        return self.start_time

    def __exit__(self, exc_type, exc_value, traceback):
        import time
        end_time = time.time()
        duration = end_time - self.start_time
        print(f"Timer {self.name}: {duration:.4f} seconds")
        return False

# Modern function-based approach (concise)
@contextmanager
def timer(name):
    """Context manager for timing operations."""
    import time
    print(f"Starting timer: {name}")
    start_time = time.time()

    try:
        yield start_time  # This value goes to 'as' variable
    finally:
        end_time = time.time()
        duration = end_time - start_time
        print(f"Timer {name}: {duration:.4f} seconds")

# Both work the same way
```

```
print("=== Class-based Context Manager ===")
with TimerClass("class_timer") as start:
    import time
    time.sleep(0.1)
    print(f"Started at: {start}")

print("\n=== Function-based Context Manager ===")
with timer("function_timer") as start:
    import time
    time.sleep(0.1)
    print(f"Started at: {start}")
```

Key Insight: The `@contextmanager` decorator converts a generator function into a context manager. Everything before `yield` is like `__enter__`, and everything after `yield` (in the `finally` block) is like `__exit__`.

Using @contextmanager decorator:

```
@contextmanager
def temporary_directory():
    """Context manager for temporary directories."""
    import tempfile
    import shutil

    print("Creating temporary directory...")
    temp_dir = tempfile.mkdtemp()
    print(f" Created: {temp_dir}")

    try:
        yield temp_dir  # Provide the directory path to the with block
    finally:
        print(f"Cleaning up temporary directory: {temp_dir}")
        try:
            shutil.rmtree(temp_dir)
            print(" Cleanup completed")
        except Exception as e:
            print(f" Cleanup failed: {e}")

# Usage
print("=== Temporary Directory Example ===")
with temporary_directory() as temp_dir:
    print(f"Working in: {temp_dir}")

    # Create some files
    temp_file = f"{temp_dir}/test.txt"
    with open(temp_file, "w") as f:
        f.write("Temporary content")

    print(f"Created file: {temp_file}")

    # List contents
```

```
    import os
    files = os.listdir(temp_dir)
    print(f"Directory contents: {files}")

print("Directory has been cleaned up!")
```

Step 2: Configuration management with context managers

```
@contextmanager
def config_override(**kwargs):
    """Context manager for temporarily overriding configuration."""
    import os

    print("Setting up configuration overrides...")

    # Store original values
    original_values = {}
    for key, value in kwargs.items():
        original_values[key] = os.environ.get(key)
        os.environ[key] = str(value)
        print(f"  Set {key} = {value}")

    try:
        yield  # No specific value needed
    finally:
        print("Restoring original configuration...")

        # Restore original values
        for key, original_value in original_values.items():
            if original_value is None:
                if key in os.environ:
                    del os.environ[key]
                    print(f"  Removed {key}")
            else:
                os.environ[key] = original_value
                print(f"  Restored {key} = {original_value}")

# Usage
import os

print("=== Configuration Override Example ===")
print(f"DEBUG before: {os.environ.get('DEBUG', 'Not set')}")
print(f"LOG_LEVEL before: {os.environ.get('LOG_LEVEL', 'Not set')}")

with config_override(DEBUG="True", LOG_LEVEL="INFO"):
    print(f"DEBUG during: {os.environ.get('DEBUG')}")
    print(f"LOG_LEVEL during: {os.environ.get('LOG_LEVEL')}")

    # Your application code here
    print("Running with debug configuration...")
```

```
print(f"DEBUG after: {os.environ.get('DEBUG', 'Not set')}")
print(f"LOG_LEVEL after: {os.environ.get('LOG_LEVEL', 'Not set')}")
```

Step 3: Exception handling patterns

```
@contextmanager
def ignore_errors(*exception_types):
    """Context manager that ignores specified exception types."""
    try:
        yield
    except exception_types as e:
        print(f"Ignoring {type(e).__name__}: {e}")
        # By not re-raising, we suppress the exception

@contextmanager
def log_and_reraise_errors(operation_name="operation"):
    """Context manager that logs errors but doesn't suppress them."""
    try:
        yield
    except Exception as e:
        print(f"Error in {operation_name}: {type(e).__name__}: {e}")
        print(f"Re-raising exception for proper handling...")
        raise  # Re-raise the exception

# Usage examples
print("\n=== Exception Handling Examples ===")

print("1. Ignoring specific errors:")
with ignore_errors(ValueError, TypeError):
    print("Attempting to convert 'abc' to int...")
    result = int("abc")  # This would normally raise ValueError
    print("This won't be reached")

print("Execution continues after ignored error!")

print("\n2. Logging errors without suppressing:")
try:
    with log_and_reraise_errors("data processing"):
        print("Processing data...")
        raise RuntimeError("Processing failed!")

except RuntimeError as e:
    print(f"Caught and handled: {e}")

print("Application continues...")
```

Step 4: Practical contextmanager examples

```python
@contextmanager
def change_directory(path):
    """Context manager to temporarily change working directory."""
    import os

    original_dir = os.getcwd()
    print(f"Changing directory from {original_dir} to {path}")

    try:
        os.chdir(path)
        yield path
    finally:
        os.chdir(original_dir)
        print(f"Restored directory to {original_dir}")

@contextmanager
def suppress_output():
    """Context manager to suppress stdout temporarily."""
    import sys
    from io import StringIO

    original_stdout = sys.stdout
    print("Suppressing output...")

    try:
        sys.stdout = StringIO()  # Redirect to nowhere
        yield
    finally:
        sys.stdout = original_stdout
        print("Output restored!")

# Usage
print("\n=== Practical Examples ===")

# Directory change example
current_dir = os.getcwd()
print(f"Currently in: {current_dir}")

with change_directory("/tmp"):
    print(f"Now in: {os.getcwd()}")
    # Do work in /tmp

print(f"Back in: {os.getcwd()}")

# Output suppression example
print("This will be visible")

with suppress_output():
    print("This will be hidden")
    print("So will this")

print("This will be visible again")
```

Commentary:

The @contextmanager decorator significantly simplifies context manager creation, making it easy to wrap any setup/teardown pattern in a reusable, readable context manager.

III. Database and Connection Management (15 min)

Mental Model: Database Connections as Hotel Rooms

Think of database connections like hotel rooms:

- **Check-in**: Get your room (establish connection)
- **Use Safely**: Use the room, handle any problems
- **Check-out**: Always return the room (close connection), even if you had problems
- **Transactions**: Like "Do Not Disturb" signs - finish everything or reset to start

Context managers are perfect for database and connection management.

Step 1: Basic database connection manager

```
import sqlite3
from contextlib import contextmanager

class DatabaseConnection:
    """Context manager for database connections."""

    def __init__(self, db_path):
        self.db_path = db_path
        self.connection = None
        print(f"DatabaseConnection created for: {db_path}")

    def __enter__(self):
        print(f"Connecting to database: {self.db_path}")
        try:
            self.connection = sqlite3.connect(self.db_path)
            print(" Database connection established")
            return self.connection
        except Exception as e:
            print(f" Failed to connect to database: {e}")
            raise

    def __exit__(self, exc_type, exc_value, traceback):
        if self.connection:
            if exc_type is None:
                print("Closing database connection normally")
            else:
                print(f"Closing database connection after exception:
{exc_type.__name__}")

            try:
                self.connection.close()
```

```
                print(" Database connection closed")
            except Exception as e:
                print(f" Error closing database: {e}")

        # Don't suppress exceptions
        return False

# Usage
print("=== Basic Database Connection ===")

# Create a test database
with DatabaseConnection(":memory:") as conn:
    cursor = conn.cursor()

    # Create table
    cursor.execute("""
        CREATE TABLE users (
            id INTEGER PRIMARY KEY,
            name TEXT NOT NULL,
            email TEXT UNIQUE
        )
    """)
    print(" Table created")

    # Insert data
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)",
                   ("Alice", "alice@example.com"))
    cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)",
                   ("Bob", "bob@example.com"))
    print(" Data inserted")

    # Query data
    cursor.execute("SELECT * FROM users")
    users = cursor.fetchall()
    print(f" Retrieved users: {users}")

print("Database connection automatically closed")
```

Step 2: Transaction management with context managers

```
@contextmanager
def database_transaction(db_path):
    """Context manager for database transactions."""
    print(f"Starting database transaction: {db_path}")

    conn = sqlite3.connect(db_path)

    try:
        # Start transaction
        conn.execute("BEGIN TRANSACTION")
        print(" Transaction started")
```

```
        yield conn

        # If we get here, commit the transaction
        conn.execute("COMMIT")
        print(" Transaction committed")

    except Exception as e:
        # If anything goes wrong, rollback
        print(f" Error in transaction: {e}")
        try:
            conn.execute("ROLLBACK")
            print(" Transaction rolled back")
        except Exception as rollback_error:
            print(f" Rollback failed: {rollback_error}")
        raise

    finally:
        conn.close()
        print(" Database connection closed")

# Usage
print("\n=== Transaction Management ===")

# Successful transaction
print("1. Successful transaction:")
with database_transaction(":memory:") as conn:
    cursor = conn.cursor()

    # Setup
    cursor.execute("CREATE TABLE accounts (id INTEGER, balance REAL)")
    cursor.execute("INSERT INTO accounts VALUES (1, 100.0)")
    cursor.execute("INSERT INTO accounts VALUES (2, 50.0)")

    # Transfer money
    cursor.execute("UPDATE accounts SET balance = balance - 25 WHERE id =
1")
    cursor.execute("UPDATE accounts SET balance = balance + 25 WHERE id =
2")

    print("Money transfer completed")

# Failed transaction
print("\n2. Failed transaction:")
try:
    with database_transaction(":memory:") as conn:
        cursor = conn.cursor()

        # Setup
        cursor.execute("CREATE TABLE accounts (id INTEGER, balance REAL)")
        cursor.execute("INSERT INTO accounts VALUES (1, 100.0)")

        # Valid operation
        cursor.execute("UPDATE accounts SET balance = balance - 25 WHERE
id = 1")
```

```
        print("Deducted money from account 1")

        # This will cause an error (simulated business logic failure)
        raise ValueError("Transfer failed due to insufficient funds!")

        # This line would never execute
        cursor.execute("UPDATE accounts SET balance = balance + 25 WHERE
id = 2")

except ValueError as e:
    print(f"Transaction failed: {e}")

print("Application continues...")
```

Step 3: Connection pooling basics

```
@contextmanager
def pooled_connection(pool, timeout=30):
    """Context manager for getting connections from a pool."""
    print("Getting connection from pool...")

    # Simulate getting connection from pool
    connection = f"pooled_connection_{id(pool)}"
    print(f" Got connection: {connection}")

    try:
        yield connection
    except Exception as e:
        print(f"Error using pooled connection: {e}")
        # Could mark connection as bad here
        raise
    finally:
        print(f"Returning connection to pool: {connection}")
        # In real implementation, would return to pool

class SimpleConnectionPool:
    """Simplified connection pool for demonstration."""

    def __init__(self, max_connections=5):
        self.max_connections = max_connections
        self.active_connections = 0
        print(f"Created connection pool (max: {max_connections})")

    @contextmanager
    def get_connection(self):
        """Get a connection from the pool."""
        if self.active_connections >= self.max_connections:
            raise RuntimeError("Pool exhausted!")

        self.active_connections += 1
        connection_id = f"conn_{self.active_connections}"
        print(f" Pool: Allocated {connection_id}
```

```
({self.active_connections}/{self.max_connections})")

        try:
            yield connection_id
        finally:
            self.active_connections -= 1
            print(f" Pool: Released {connection_id}
({self.active_connections}/{self.max_connections})")

# Usage
print("\n=== Connection Pooling ===")

pool = SimpleConnectionPool(max_connections=2)

def do_database_work(work_id):
    """Simulate database work using pooled connection."""
    with pool.get_connection() as conn:
        print(f"Work {work_id}: Using {conn}")
        import time
        time.sleep(0.1)  # Simulate work
        print(f"Work {work_id}: Completed")

# Multiple workers using the pool
do_database_work(1)
do_database_work(2)

print("Pool usage completed")
```

Commentary:

Context managers provide elegant solutions for database connection management, ensuring connections are always closed and transactions are properly handled, even when errors occur.

IV. Practical Applications (5 min)

Let's look at some common real-world applications of context managers.

File operations with context managers:

```
@contextmanager
def safe_file_write(filename, backup=True):
    """Safely write to a file with automatic backup and error recovery."""
    import os
    import shutil

    backup_file = f"{filename}.backup" if backup else None
    temp_file = f"{filename}.tmp"

    # Create backup if file exists
```

```
    if backup and os.path.exists(filename):
        shutil.copy2(filename, backup_file)
        print(f" Created backup: {backup_file}")

    # Open temporary file for writing
    file_handle = open(temp_file, 'w')
    print(f" Opened temporary file: {temp_file}")

    try:
        yield file_handle

        # If we get here, writing was successful
        file_handle.close()

        # Replace original with temporary
        if os.path.exists(filename):
            os.remove(filename)
        os.rename(temp_file, filename)
        print(f" Successfully wrote to: {filename}")

        # Remove backup on success
        if backup_file and os.path.exists(backup_file):
            os.remove(backup_file)
            print(f" Removed backup file")

    except Exception as e:
        print(f" Error during write: {e}")

        # Close file handle if still open
        if not file_handle.closed:
            file_handle.close()

        # Remove temporary file
        if os.path.exists(temp_file):
            os.remove(temp_file)
            print(f" Cleaned up temporary file")

        # Restore from backup if needed
        if backup_file and os.path.exists(backup_file):
            if os.path.exists(filename):
                os.remove(filename)
            os.rename(backup_file, filename)
            print(f" Restored from backup")

        raise

# Usage
print("=== Safe File Writing ===")

# Create initial file
with open("important.txt", "w") as f:
    f.write("Original important data\n")

# Successful write
```

```
print("1. Successful write:")
with safe_file_write("important.txt") as f:
    f.write("Updated important data\n")
    f.write("More updates\n")

# Check result
with open("important.txt", "r") as f:
    print(f"File contents: {f.read().strip()}")

# Simulate failed write
print("\n2. Failed write (will be recovered):")
try:
    with safe_file_write("important.txt") as f:
        f.write("This write will fail\n")
        raise IOError("Simulated write error!")

except IOError as e:
    print(f"Write failed: {e}")

# Check that file was restored
with open("important.txt", "r") as f:
    print(f"File contents after failed write: {f.read().strip()}")

# Cleanup
import os
if os.path.exists("important.txt"):
    os.remove("important.txt")
```

Commentary:

Context managers excel at wrapping complex operations that require guaranteed cleanup, making error-prone operations safe and reliable.

V. Recap & Best Practices (5 min)

Key Takeaways:

- Context managers guarantee resource cleanup through the `with` statement.
- The `__enter__` and `__exit__` methods define the context manager protocol.
- `@contextmanager` decorator simplifies creation of context managers using generator functions.
- Context managers are essential for file operations, database connections, and configuration management.
- Proper exception handling in context managers prevents resource leaks.

Best Practices:

- Always implement proper exception handling in `__exit__` methods.
- Use `@contextmanager` for simple setup/teardown patterns.
- Document whether your context manager suppresses exceptions.
- Use context managers for any resource that needs guaranteed cleanup.

- Test context managers with both successful and error scenarios.

When to Use Context Managers:

- File and database operations
- Temporary configuration changes
- Resource allocation (connections, locks, etc.)
- Setup/teardown for testing
- Timing and performance monitoring
- Any operation requiring guaranteed cleanup

Common Patterns:

- Resource acquisition and release
- Configuration override and restore
- Exception logging and handling
- Temporary state changes
- Safe file operations with backup/recovery

Final Multiple-Choice Question:

What is the main advantage of using context managers for resource management?

A. They make code run faster B. They guarantee cleanup even when exceptions occur C. They reduce memory usage D. They automatically handle all errors

(Answer: B. They guarantee cleanup even when exceptions occur - this is the primary purpose of context managers, ensuring resources are properly released regardless of how the code block exits.)