Advanced Python Workbook: Decorators and Context Managers

This workbook is designed for intermediate Python developers transitioning to advanced concepts. It provides a deep dive into **Decorators** and **Context Managers**, focusing on idiomatic, functional implementation, error handling, and practical applications.

1️⃣ Decorators – Basics

This section focuses on the fundamental concepts: creating a simple decorator, understanding function wrapping, and preserving metadata.

Simple

Exercise 1.1: Simple Greeting Decorator Create a decorator `add_greeting` that prints "Hello!" before executing the decorated function.

```
def add_greeting(func):
    # SIMPLE DECORATOR TEMPLATE
    def wrapper():
        print("Hello!")
        ???
    return wrapper

@add_greeting
def say_name():
    print("My name is Alex.")

say_name()
```

Expected Output:

```
Hello!
My name is Alex.
```

Exercise 1.2: Function Alias Checker Create a decorator `check_alias` that prints the name of the function it decorates before executing it.

```
def check_alias(func):
    def wrapper():
        print(f"Executing function: {???}")
        func()
    return wrapper

@check_alias
def calculate_sum():
    print(10 + 20)
```

```
calculate_sum()
```

Expected Output (Example):

```
Executing function: calculate_sum
30
```

Exercise 1.3: Post-Execution Message Create a decorator `post_message` that executes the decorated function first, then prints a fixed "Execution complete." message afterwards.

```
def post_message(func):
    def wrapper():
        ???
        print("Execution complete.")
    return wrapper

@post_message
def cleanup():
    print("Cleaning up resources.")

cleanup()
```

Expected Output:

```
Cleaning up resources.
Execution complete.
```

Medium

Exercise 1.4: Decorator for *args and **kwargs Create a decorator `accept_everything` that can wrap any function, regardless of its arguments. The decorator should print a message showing all positional and keyword arguments passed to the function before calling it.

```
def accept_everything(func):
    def wrapper(???, ???):
        print(f"Args: {args}")
        print(f"Kwargs: {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@accept_everything
def describe_item(name, price, stock=True):
    return f"Item: {name}, Price: ${price}, In Stock: {stock}"

result = describe_item("Laptop", 1200, stock=False)
```

```
print(result)
```

Expected Output:

```
Args: ('Laptop', 1200)
Kwargs: {'stock': False}
Item: Laptop, Price: $1200, In Stock: False
```

Exercise 1.5: Simple Return Value Wrapper Create a decorator `double_result` that takes a function that returns a number and modifies the returned value by multiplying it by 2.

```
def double_result(func):
    def wrapper(*args, **kwargs):
        original_result = ???
        return original_result * ???
    return wrapper

@double_result
def get_number(value):
    return value

print(get_number(10))
```

Expected Output:

```
20
```

Hard

Exercise 1.6: The `functools.wraps` Requirement Decorators hide the original function's name and metadata. Create two versions of a decorator: `debug_no_wraps` and `debug_with_wraps`. Both decorators should print "DEBUG: Running function." Demonstrate the difference in their output when checking the `__name__` and `__doc__` attributes of the decorated function.

```
from functools import wraps

def debug_no_wraps(func):
    def wrapper(*args, **kwargs):
        print("DEBUG: Running function.")
        return func(*args, **kwargs)
    return wrapper

def debug_with_wraps(func):
    def wrapper(*args, **kwargs):
        print("DEBUG: Running function.")
        return func(*args, **kwargs)
```

```
    return ??? # Use wraps here

@debug_no_wraps
def compute_data_a():
    """Calculates data A."""
    return 1

@debug_with_wraps
def compute_data_b():
    """Calculates data B."""
    return 2

# Check metadata
print(f"A Name (No Wraps): {compute_data_a.__name__}")
print(f"B Name (With Wraps): {compute_data_b.__name__}")
```

Expected Behavior: The output for `compute_data_a.__name__` should be the name of the **wrapper function**, while the output for `compute_data_b.__name__` should correctly show `compute_data_b`.

Exercise 1.7: Conditional Execution with Decorator Create a decorator `run_if_positive` that accepts one positional argument (assumed to be the first argument of the decorated function). The decorated function should only be executed if this argument is greater than 0. Otherwise, the function should return `None` and print an error message.

```
def run_if_positive(func):
    def wrapper(number, *args, **kwargs):
        if number > ???:
            return func(number, *args, **kwargs)
        else:
            print(f"ERROR: Cannot run {func.__name__} with non-positive input: {n
            return ???
    return wrapper

@run_if_positive
def safe_divide(x, y):
    return x / y

print(safe_divide(5, 2))
print(safe_divide(-1, 5))
```

Expected Output:

```
2.5
ERROR: Cannot run safe_divide with non-positive input: -1
None
```

2 Decorators – Advanced Patterns

This section focuses on practical decorator use cases: timing, logging, and validation.

Simple

Exercise 2.1: Simple Function Logger Create a decorator `log_call` that prints the function's name and the values of its arguments *before* execution. Assume the function takes two arguments.

```
from functools import wraps

def log_call(func):
    @wraps(func)
    def wrapper(a, b):
        print(f"LOG: Calling {???} with args: ({a}, {b})")
        return func(a, b)
    return wrapper

@log_call
def multiply(x, y):
    return x * y

print(multiply(8, 7))
```

Expected Output:

```
LOG: Calling multiply with args: (8, 7)
56
```

Exercise 2.2: Execution Timer (Start/End) Create a decorator `timing_start_end` that prints a "START" message before the function and an "END" message after the function. (No actual time calculation needed yet).

```
from functools import wraps

def timing_start_end(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"--- START: {func.__name__}")
        result = ???
        print(f"--- END: {func.__name__}")
        return result
    return wrapper

@timing_start_end
def process_data():
    return [x * 2 for x in range(3)]

print(process_data())
```

Expected Output:

```
--- START: process_data
--- END: process_data
[0, 2, 4]
```

Medium

Exercise 2.3: Type Validation Decorator Create a decorator `validate_integer` that checks if the first positional argument passed to the function is an integer. If it is not, the decorator should raise a `TypeError` with a descriptive message.

```
from functools import wraps

def validate_integer(func):
    @wraps(func)
    def wrapper(value, *args, **kwargs):
        if not isinstance(value, ???):
            raise ???(f"Input must be an integer, got {type(value).__name__}")
        return func(value, *args, **kwargs)
    return wrapper

@validate_integer
def process_id(user_id):
    return f"Processing ID: {user_id}"

# Test cases
print(process_id(123))
# process_id("abc") # This should raise an error
```

Expected Behavior: The first call prints the string. The second call (if uncommented) raises `TypeError: Input must be an integer, got str`.

Exercise 2.4: Log Input and Output Enhance the logging decorator to also print the returned value of the function.

```
from functools import wraps

def log_io(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"IO LOG: {func.__name__} called with args: {args}, kwargs: {kwargs
        result = func(*args, **kwargs)
        print(f"IO LOG: {func.__name__} returned: {???}")
        return result
    return wrapper

@log_io
def power(base, exp=2):
    return base ** exp
```

```
print(power(3, exp=3))
```

Expected Output:

```
IO LOG: power called with args: (3,), kwargs: {'exp': 3}
IO LOG: power returned: 27
27
```

Exercise 2.5: Simple Caching (No arguments) Create a simple caching decorator `simple_cache` that stores the result of a function call in a module-level dictionary (the cache). If the function is called again, it should return the cached value and print "From cache." Assume the function takes no arguments and always returns the same value.

```
from functools import wraps

CACHE = {}

def simple_cache(func):
    @wraps(func)
    def wrapper():
        if func.__name__ in CACHE:
            print("From cache.")
            return ???
        else:
            result = func()
            CACHE[func.__name__] = result
            return result
    return wrapper

@simple_cache
def expensive_call():
    print("Performing expensive calculation...")
    return 42

print(expensive_call())
print(expensive_call())
```

Expected Output:

```
Performing expensive calculation...
42
From cache.
42
```

Hard

Exercise 2.6: Access Control / Authorization Decorator Create a decorator `requires_admin` that checks if a global variable `CURRENT_USER_ROLE` is set to "admin". If the user is not an admin, it should raise a `PermissionError`.

```
from functools import wraps

CURRENT_USER_ROLE = "viewer"

def requires_admin(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        global CURRENT_USER_ROLE
        if CURRENT_USER_ROLE == ???:
            return func(*args, **kwargs)
        else:
            raise PermissionError(f"User '{CURRENT_USER_ROLE}' is not authorized
    return wrapper

@requires_admin
def delete_database():
    return "Database deleted successfully."

# Test the function with current role
# delete_database() # This should raise an error

# Change role and test again
CURRENT_USER_ROLE = "admin"
print(delete_database())
```

Expected Behavior: The first call (if uncommented) raises `PermissionError`. The second call prints the success message.

Exercise 2.7: Advanced Caching with Args and Kwargs Refactor the caching decorator to handle functions with arguments. The key for the cache should be a combination of the function name, positional arguments, and keyword arguments (converted to a tuple for hashability).

```
from functools import wraps

ADVANCED_CACHE = {}

def smart_cache(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Create a unique, hashable key from function name, args, and sorted kwar
        key_parts = [func.__name__, args, tuple(sorted(kwargs.items()))]
        cache_key = tuple(key_parts)

        if cache_key in ADVANCED_CACHE:
            print("From advanced cache.")
            return ADVANCED_CACHE[cache_key]
        else:
            result = func(*args, **kwargs)
            ADVANCED_CACHE[cache_key] = result
            return result
```

```
    return wrapper

@smart_cache
def fibonacci(n):
    print(f"Calculating fibonacci({n})...")
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(4))
print(fibonacci(4))
print(fibonacci(3))
```

Expected Behavior: `fibonacci(4)` calculation should only happen once. `fibonacci(3)` should calculate separately.

Exercise 2.8: Execution Timer (Actual Time) Use the `time` module (which is available in the Python environment) to create a decorator `time_it` that measures and prints the actual execution time of a decorated function in seconds.

```
from functools import wraps
import time

def time_it(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = ???
        result = func(*args, **kwargs)
        end_time = ???
        duration = end_time - start_time
        print(f"Time: {func.__name__} took {duration:.4f} seconds.")
        return result
    return wrapper

@time_it
def delay_task(seconds):
    time.sleep(seconds)
    return f"Slept for {seconds}s"

print(delay_task(0.05))
```

Expected Behavior: The output should show a duration of approximately 0.0500 seconds, and then the function's return value.

3️⃣ Decorators with Arguments & Stacking

This section explores creating decorator factories (decorators that accept arguments) and stacking multiple decorators on a single function.

Simple

Exercise 3.1: Simple Prefix Decorator Factory Create a decorator factory `add_prefix(prefix)` that accepts a string `prefix` as an argument. The resulting decorator should prepend that prefix to the function's printed output.

```
from functools import wraps

def add_prefix(prefix): # Decorator Factory
    def decorator(func): # Actual Decorator
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(f"[{prefix}]")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@add_prefix(???)
def report_status():
    print("System is nominal.")

report_status()
```

Expected Output:

```
[STATUS]
System is nominal.
```

Exercise 3.2: Times-to-Run Factory Create a decorator factory `run_n_times(n)` that accepts an integer `n`. The decorated function should execute exactly `n` times.

```
from functools import wraps

def run_n_times(n):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for i in range(???):
                print(f"Run {i+1}/{n}: ", end="")
                func(*args, **kwargs)
        return wrapper
    return decorator

@run_n_times(3)
def blast_off():
    print("T-minus 10...")

blast_off()
```

Expected Output:

```
Run 1/3: T-minus 10...
Run 2/3: T-minus 10...
Run 3/3: T-minus 10...
```

Exercise 3.3: Basic Decorator Stacking Apply the `add_greeting` decorator (from 1.1) and the `post_message` decorator (from 1.3) to a single function. Observe the order of execution.

```
# Assume add_greeting and post_message are defined as in 1.1 and 1.3

def add_greeting(func):
    def wrapper():
        print("Hello!")
        func()
    return wrapper

def post_message(func):
    def wrapper():
        func()
        print("Execution complete.")
    return wrapper

@post_message # Outer/First-applied decorator
@add_greeting # Inner/Second-applied decorator
def perform_task():
    print("Performing task...")

perform_task()
```

Expected Output: (Trace the order carefully: the inner decorator wraps the function, the outer decorator wraps the inner decorator's wrapper)

```
Hello!
Performing task...
Execution complete.
```

Medium

Exercise 3.4: Logging with Level Control (Factory) Create a decorator factory `log_with_level(level)` that accepts a logging level (e.g., "INFO", "WARN"). The decorator should only log the function execution if the global `LOG_THRESHOLD` is equal to or below the provided level.

```
from functools import wraps

LOG_ORDER = {"DEBUG": 1, "INFO": 2, "WARN": 3, "ERROR": 4}
LOG_THRESHOLD = "INFO" # Only INFO and ERROR will run

def log_with_level(level):
    def decorator(func):
```

```
        @wraps(func)
        def wrapper(*args, **kwargs):
            if LOG_ORDER.get(level, 5) >= LOG_ORDER.get(LOG_THRESHOLD, 5):
                print(f"[{level}] {func.__name__} called.")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@log_with_level("INFO")
def status_check():
    return "Status OK"

@log_with_level("DEBUG")
def detailed_log():
    return "Detailed data"

print(status_check())
print(detailed_log())
```

Expected Output:

```
[INFO] status_check called.
Status OK
Detailed data
```

Exercise 3.5: Argument Validation Factory Create a decorator factory `validate_type(arg_name, expected_type)` that ensures a specific keyword argument passed to the function matches the `expected_type`. If not, raise a `TypeError`.

```
from functools import wraps

def validate_type(arg_name, expected_type):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if arg_name in kwargs and not isinstance(kwargs[arg_name], expected_t
                raise TypeError(f"Argument '{arg_name}' must be of type {expected
            # Handle positional arguments (requires inspection or careful argumen
            return func(*args, **kwargs)
        return wrapper
    return decorator

@validate_type(arg_name="limit", expected_type=int)
def fetch_records(query, limit=10):
    return f"Fetching '{query}' with limit {limit}"

print(fetch_records("users", limit=50))
# fetch_records("products", limit="all") # This should raise an error
```

Expected Behavior: The first call prints the string. The second call (if uncommented) raises `TypeError`.

Exercise 3.6: Stacking & Argument Interaction Stack two decorator factories: `add_prefix("OUTER")` and `add_prefix("INNER")`. Observe which prefix appears first.

```
from functools import wraps

def add_prefix(prefix):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(f"[{prefix}]", end=" ")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@add_prefix(???)
@add_prefix(???)
def process_data():
    print("Data processed.")

process_data()
```

Expected Output:

```
[OUTER] [INNER] Data processed.
```

Hard

Exercise 3.7: Retry Logic with Factory Create a decorator factory `retry(max_attempts, delay=0)` that retries the decorated function up to `max_attempts` times if it raises an `Exception`. Print a warning before each retry. Use `time.sleep(delay)` for the delay.

```
from functools import wraps
import time

attempt_count = 0 # Global counter for demonstration

def retry(max_attempts, delay=0):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            global attempt_count
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts:
                        print(f"ERROR: Max retries ({max_attempts}) reached. Rerc
                        raise
                    print(f"WARNING: Exception '{e}' occurred. Retrying in {delay
                    time.sleep(delay)
                    attempt_count += 1
        return wrapper
```

```
    return decorator

@retry(max_attempts=3, delay=0.01)
def unstable_network_call():
    global attempt_count
    if attempt_count < 2: # Fail on the first two runs
        raise IOError("Connection lost.")
    return "Success after retry."

print(unstable_network_call())
```

Expected Output: (The delay might not be noticeable, but the message sequence is key)

```
WARNING: Exception 'Connection lost.' occurred. Retrying in 0.01s (Attempt 1/3).
WARNING: Exception 'Connection lost.' occurred. Retrying in 0.01s (Attempt 2/3).
Success after retry.
```

Exercise 3.8: Parameter-based Caching Factory Refine the caching decorator factory `cache_for_args(cache_size)` to allow the user to specify the maximum number of unique results to store. If the cache is full and a new key is added, the oldest entry must be removed (LIFO/Simple approach).

(Note: Use a list of keys and a dictionary for the cache. Removing the oldest means removing the first element of the list/dictionary key)

```
from functools import wraps

GLOBAL_CACHE = {}
KEY_ORDER = [] # Tracks insertion order for LIFO eviction

def cache_for_args(cache_size):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            key_parts = [func.__name__, args, tuple(sorted(kwargs.items()))]
            cache_key = tuple(key_parts)

            if cache_key in GLOBAL_CACHE:
                return GLOBAL_CACHE[cache_key]

            # Cache miss - perform computation
            result = func(*args, **kwargs)

            # Check for eviction
            if len(GLOBAL_CACHE) >= ???:
                oldest_key = KEY_ORDER.pop(0) # Remove oldest key
                del GLOBAL_CACHE[oldest_key]
                print(f"CACHE: Evicting {oldest_key[0]} to make space.")

            # Store new result
            GLOBAL_CACHE[cache_key] = result
            KEY_ORDER.append(cache_key)
```

```
            return result
        return wrapper
    return decorator

@cache_for_args(cache_size=2)
def add(a, b):
    print(f"Calculating {a} + {b}...")
    return a + b

print(add(1, 1)) # Key 1: calculated
print(add(2, 2)) # Key 2: calculated
print(add(3, 3)) # Key 3: calculated (evicts Key 1)
print(add(1, 1)) # Key 1: Should calculate again after eviction
```

Expected Output (The key is the eviction):

```
Calculating 1 + 1...
2
Calculating 2 + 2...
4
Calculating 3 + 3...
CACHE: Evicting add to make space.
6
Calculating 1 + 1...
2
```

Exercise 3.9: Sequential Validator Stacking Stack three validation decorators (`validate_int`, `validate_range(min_val, max_val)`, `validate_positive`) on a single function that checks if an input is an integer, between 10 and 20, and positive.

```
from functools import wraps

# Assume a basic validate_int (checks if arg is an int) is available
def validate_int(func):
    @wraps(func)
    def wrapper(value):
        if not isinstance(value, int):
            raise TypeError("Value must be an integer.")
        return func(value)
    return wrapper

# Decorator Factory 1: Range Check
def validate_range(min_val, max_val):
    def decorator(func):
        @wraps(func)
        def wrapper(value):
            if not (min_val <= value <= max_val):
                raise ValueError(f"Value must be between {min_val} and {max_val}.
            return func(value)
        return wrapper
    return decorator

# Decorator 2: Positive Check
def validate_positive(func):
```

```
    @wraps(func)
    def wrapper(value):
        if value <= 0:
            raise ValueError("Value must be positive.")
        return func(value)
    return wrapper

@validate_range(???, ???)
@validate_positive
@validate_int
def process_score(score):
    return f"Score {score} is valid."

print(process_score(15))
# process_score(5) # Fails positive check
# process_score(25) # Fails range check
# process_score(15.5) # Fails int check
```

Expected Behavior: `process_score(15)` prints the success message. The subsequent commented calls should raise exceptions based on the order of execution (inner to outer).

4️⃣ Context Managers – Basics

This section introduces the concept of resource setup and teardown using the `with` statement, focusing on the functional approach with `contextlib.contextmanager`.

Simple

Exercise 4.1: Simple Greeting Context Manager Use `contextlib.contextmanager` to create a generator-based context manager `greet_context` that prints "Entering block" on entry and "Exiting block" on exit.

```
from contextlib import contextmanager

@contextmanager
def greet_context():
    print(???) # Setup
    try:
        yield # Execution
    finally:
        print(???) # Teardown

with greet_context():
    print("Inside the block.")
```

Expected Output:

```
Entering block
Inside the block.
Exiting block
```

Exercise 4.2: File-like Connection Mock Create a context manager `mock_connection(name)` that prints "Connecting to [name]..." on entry and "Closing connection to [name]." on exit.

```
from contextlib import contextmanager

@contextmanager
def mock_connection(name):
    print(f"Connecting to {name}...")
    try:
        yield ??? # The execution block
    finally:
        print(f"Closing connection to {name}.")

with mock_connection("DB_A"):
    print("Performing queries...")
```

Expected Output:

```
Connecting to DB_A...
Performing queries...
Closing connection to DB_A.
```

Exercise 4.3: State Change Context Manager Create a context manager `temp_state(variable_name, temp_value)` that temporarily sets a global variable to `temp_value` on entry, and restores the original value on exit.

```
from contextlib import contextmanager

GLOBAL_SETTING = "DEFAULT"

@contextmanager
def temp_state(variable_name, temp_value):
    global GLOBAL_SETTING
    original_value = GLOBAL_SETTING
    print(f"Old setting: {original_value}")
    GLOBAL_SETTING = temp_value
    try:
        yield
    finally:
        GLOBAL_SETTING = original_value
        print(f"Setting restored: {GLOBAL_SETTING}")

with temp_state("GLOBAL_SETTING", "TEST_MODE"):
    print(f"Inside with: {GLOBAL_SETTING}")

print(f"Outside with: {GLOBAL_SETTING}")
```

Expected Output:

```
Old setting: DEFAULT
Inside with: TEST_MODE
Setting restored: DEFAULT
Outside with: DEFAULT
```

Medium

Exercise 4.4: Simple Lock Manager Using `threading.Lock` (imported from `threading`), create a context manager `lock_manager(lock)` that acquires the lock on entry and releases it on exit.

```
from contextlib import contextmanager
import threading

# Global lock instance
my_lock = threading.Lock()

@contextmanager
def lock_manager(lock):
    print("Attempting to acquire lock...")
    lock.acquire()
    try:
        yield
    finally:
        lock.release()
        print("Lock released.")

with lock_manager(my_lock):
    print("Lock acquired, critical section running.")
    assert my_lock.locked() == ??? # Should be True

assert my_lock.locked() == ??? # Should be False
```

Expected Output:

```
Attempting to acquire lock...
Lock acquired, critical section running.
True
Lock released.
False
```

Exercise 4.5: Returning a Value from the Context Modify the `mock_connection` context manager to yield the connection name (`name`) so it can be captured by the `as` clause in the `with` statement.

```
from contextlib import contextmanager

@contextmanager
def mock_connection(name):
```

```
    print(f"Connecting to {name}...")
    try:
        yield name # Yield the connection object/resource
    finally:
        print(f"Closing connection to {name}.")

with mock_connection("DB_B") as conn:
    print(f"Using connection: {conn}")
    assert conn == ???
```

Expected Output:

```
Connecting to DB_B...
Using connection: DB_B
DB_B
Closing connection to DB_B.
```

Hard

Exercise 4.6: Nested Context Managers (Observation) Nest two instances of the `greet_context` from 4.1. Predict and observe the order of entry and exit messages.

```
from contextlib import contextmanager

@contextmanager
def greet_context(tag):
    print(f"Entering block: {tag}")
    try:
        yield
    finally:
        print(f"Exiting block: {tag}")

with greet_context("Outer"):
    print("Outer action.")
    with greet_context("Inner"):
        print("Inner action.")
    print("Back in Outer.")
```

Expected Output: The entry messages should be **Outer** then **Inner**, and the exit messages should be **Inner** then **Outer**.

Exercise 4.7: Tracking Usage Count Create a context manager `usage_tracker(counter)` that increments a global counter on entry and decrements it on exit, ensuring the counter is 0 when all blocks are done.

```
from contextlib import contextmanager

GLOBAL_COUNT = 0

@contextmanager
```

```
def usage_tracker(name):
    global GLOBAL_COUNT
    GLOBAL_COUNT += 1
    print(f"[{name}] Entered. Count: {GLOBAL_COUNT}")
    try:
        yield
    finally:
        GLOBAL_COUNT -= 1
        print(f"[{name}] Exited. Count: {GLOBAL_COUNT}")

with usage_tracker("A"):
    with usage_tracker("B"):
        pass

assert GLOBAL_COUNT == 0
```

Expected Output:

```
[A] Entered. Count: 1
[B] Entered. Count: 2
[B] Exited. Count: 1
[A] Exited. Count: 0
```

5️⃣ Context Managers – Advanced & Exception Handling

This section explores how context managers interact with exceptions and how to suppress or handle errors during the teardown phase.

Simple

Exercise 5.1: Observing Exception Flow Create a context manager `exception_observer` that has no logic in its body. Place a `try...except` block outside the `with` statement to catch an exception raised inside the block. Observe that the exception handler outside the `with` runs.

```
from contextlib import contextmanager

@contextmanager
def exception_observer():
    print("Observer: Setup.")
    try:
        yield
    finally:
        print("Observer: Teardown.")

try:
    with exception_observer():
        print("Raising error.")
        raise ValueError("Simulated failure.")
except ValueError as e:
    print(f"Caught exception outside: {e}")
```

Expected Output:

```
Observer: Setup.
Raising error.
Observer: Teardown.
Caught exception outside: Simulated failure.
```

Exercise 5.2: Suppressing Specific Exceptions Create a context manager `error_suppressor(error_type)` that accepts an error type (e.g., `ZeroDivisionError`). If that specific error occurs inside the `with` block, the context manager should prevent it from propagating, printing a suppression message instead. Other errors must propagate.

```
from contextlib import contextmanager

@contextmanager
def error_suppressor(error_type):
    print("Suppressor active.")
    try:
        yield
    except error_type as e:
        print(f"SUPPRESSED: {e.__class__.__name__} was ignored.")
    except Exception as e:
        # Re-raise anything else
        raise e
    finally:
        print("Suppressor deactivated.")

with error_suppressor(ZeroDivisionError):
    print(10 / 0) # This should be suppressed

try:
    with error_suppressor(ZeroDivisionError):
        print("No error yet.")
        raise IndexError("Should not be suppressed.")
except IndexError as e:
    print(f"Caught non-suppressed error: {e}")
```

Expected Output:

```
Suppressor active.
SUPPRESSED: ZeroDivisionError was ignored.
Suppressor deactivated.
Suppressor active.
No error yet.
Suppressor deactivated.
Caught non-suppressed error: Should not be suppressed.
```

Medium

Exercise 5.3: Simple Block Timer CM Create a generator-based context manager `timer_cm` that uses `time.time()` to measure and print the duration of the enclosed code block.

```
from contextlib import contextmanager
import time

@contextmanager
def timer_cm():
    start_time = ???
    try:
        yield
    finally:
        end_time = ???
        duration = end_time - start_time
        print(f"Block executed in {duration:.4f} seconds.")

with timer_cm():
    time.sleep(0.02)
    print("Operation done.")
```

Expected Behavior: The output should show a duration of approximately 0.0200 seconds.

Exercise 5.4: Custom Exception Propagation Create a context manager `convert_error` that catches any `ValueError` raised inside the block and converts it into a `TypeError` before propagating it.

```
from contextlib import contextmanager

@contextmanager
def convert_error():
    try:
        yield
    except ValueError as e:
        raise TypeError(f"Value error converted: {e}")
    # Other exceptions are automatically propagated

try:
    with convert_error():
        print("Converting...")
        raise ValueError("Bad input value.")
except TypeError as e:
    print(f"Successfully caught converted error: {e}")
except ValueError as e:
    print("Error: ValueError was not converted.")
```

Expected Output:

```
Converting...
Successfully caught converted error: Value error converted: Bad input value.
```

Exercise 5.5: Clean-up After Exception Create a context manager `always_cleanup` that prints a clean-up message regardless of whether an exception occurred inside the `with` block. Include an exception inside the block to confirm the `finally` (or equivalent) section runs.

```
from contextlib import contextmanager

@contextmanager
def always_cleanup():
    print("Resource allocated.")
    try:
        yield
    finally:
        print("Resource always de-allocated.")

try:
    with always_cleanup():
        print("Working...")
        1 / 0
except ZeroDivisionError:
    print("Exception handled outside.")
```

Expected Output:

```
Resource allocated.
Working...
Resource always de-allocated.
Exception handled outside.
```

Hard

Exercise 5.6: Nested Exception Propagation and Suppression Nest two context managers: `suppress_index_error` (suppresses `IndexError`) inside `convert_error_to_key` (converts `ValueError` to `KeyError`). Raise an `IndexError`, a `ValueError`, and a `TypeError` sequentially in three separate `with` blocks and trace the outcome.

```
from contextlib import contextmanager

@contextmanager
def convert_error_to_key():
    try:
        yield
    except ValueError as e:
        raise KeyError(f"Conversion: {e}")

@contextmanager
def suppress_index_error():
    try:
        yield
    except IndexError:
        print("INDEX ERROR SUPPRESSED.")
```

```
# Test 1: IndexError (Inner CM suppresses)
with convert_error_to_key():
    with suppress_index_error():
        try:
            print("Test 1: Raising IndexError...")
            raise IndexError("Oops")
        except:
            pass # Suppressed by inner CM

# Test 2: ValueError (Inner CM ignores, Outer CM converts)
try:
    with convert_error_to_key():
        with suppress_index_error():
            print("Test 2: Raising ValueError...")
            raise ValueError("Bad value")
except KeyError as e:
    print(f"Test 2: Caught converted error: {e.__class__.__name__}")

# Test 3: TypeError (Neither CM handles)
try:
    with convert_error_to_key():
        with suppress_index_error():
            print("Test 3: Raising TypeError...")
            raise TypeError("Wrong type")
except TypeError as e:
    print(f"Test 3: Caught propagated error: {e.__class__.__name__}")
```

Expected Output:

```
Test 1: Raising IndexError...
INDEX ERROR SUPPRESSED.
Test 2: Raising ValueError...
Test 2: Caught converted error: KeyError
Test 3: Raising TypeError...
Test 3: Caught propagated error: TypeError
```

Exercise 5.7: Resource Pool Management (Global State) Create a context manager factory `pool_manager(resource_name, limit)` that uses a global list `RESOURCE_POOL` to track active resources. On entry, it checks if `limit` is reached. If full, it raises a `ResourceWarning` (or generic `Exception` if `ResourceWarning` is unavailable/tricky). On exit, it removes the resource.

```
from contextlib import contextmanager

RESOURCE_POOL = []

@contextmanager
def pool_manager(resource_name, limit):
    global RESOURCE_POOL
    if len(RESOURCE_POOL) >= ???:
        raise Exception(f"Resource pool full. Cannot allocate {resource_name}.")

    RESOURCE_POOL.append(resource_name)
```

```
    print(f"Allocated {resource_name}. Pool size: {len(RESOURCE_POOL)}")
    try:
        yield resource_name
    finally:
        RESOURCE_POOL.remove(resource_name)
        print(f"De-allocated {resource_name}. Pool size: {len(RESOURCE_POOL)}")

# Test 1: Successful allocation
with pool_manager("Connection 1", 2):
    pass

# Test 2: Allocation and full check
try:
    with pool_manager("C2", 1): # Limit is 1
        with pool_manager("C3", 1): # Fails here
            pass
except Exception as e:
    print(f"Caught error: {e}")
```

Expected Output:

```
Allocated Connection 1. Pool size: 1
De-allocated Connection 1. Pool size: 0
Allocated C2. Pool size: 1
Caught error: Resource pool full. Cannot allocate C3.
De-allocated C2. Pool size: 0
```

Exercise 5.8: Safe Threading with Context Manager Use `threading.Lock` and the `contextlib.contextmanager` pattern to implement a simplified version of `threading.Lock`'s context manager functionality. Ensure the lock is always released, even if the code block inside raises an exception.

```
from contextlib import contextmanager
import threading

my_thread_lock = threading.Lock()

@contextmanager
def safe_lock(lock):
    print("Lock acquiring...")
    lock.acquire()
    try:
        yield
    except Exception as e:
        # Important: The lock must be released even if an exception occurs
        raise e
    finally:
        lock.release()
        print("Lock released.")

try:
    with safe_lock(my_thread_lock):
        print("Critical operation started.")
```

```
        1 / 0 # Force an exception
except ZeroDivisionError:
    print("ZeroDivisionError caught outside.")

# Check lock state
print(f"Lock still held? {my_thread_lock.locked()}")
```

Expected Output:

```
Lock acquiring...
Critical operation started.
Lock released.
ZeroDivisionError caught outside.
Lock still held? False
```

6 Generator-based Context Managers (contextlib)

This section focuses entirely on the functional, generator-based approach using `@contextmanager`, reinforcing its elegance and utility.

Simple

Exercise 6.1: Context Manager for Printing Indentation Create a context manager `indent_print` that, on entry, prepends two spaces to a global `INDENT_PREFIX` variable, and on exit, removes them.

```
from contextlib import contextmanager

INDENT_PREFIX = ""

@contextmanager
def indent_print():
    global INDENT_PREFIX
    original_prefix = INDENT_PREFIX
    INDENT_PREFIX = INDENT_PREFIX + "  "
    try:
        yield
    finally:
        INDENT_PREFIX = original_prefix

def custom_print(message):
    print(f"{INDENT_PREFIX}{message}")

with indent_print():
    custom_print("Level 1")
    with indent_print():
        custom_print("Level 2")
    custom_print("Back to Level 1")
```

Expected Output:

```
  Level 1
    Level 2
  Back to Level 1
```

Exercise 6.2: Custom `open`-like Manager Create a context manager `read_data(filename)` that *mocks* a file opening process (prints messages) and yields the filename.

```
from contextlib import contextmanager

@contextmanager
def read_data(filename):
    print(f"Opening file: {filename}")
    try:
        yield ???
    finally:
        print(f"Closing file: {filename}")

with read_data("config.ini") as f:
    print(f"Processing data from {f}")
```

Expected Output:

```
Opening file: config.ini
Processing data from config.ini
Closing file: config.ini
```

Medium

Exercise 6.3: Context Manager Factory for Retries Create a context manager factory `retry_on_fail(max_attempts)` that wraps the block of code. If an exception occurs, the context manager should print a message and allow the `with` block to be re-entered (conceptually), up to `max_attempts`.

(Note: Since generator context managers are single-use, this exercise focuses on printing the state/attempt on failure, mimicking the retry logic by counting how many attempts would have been left.)

```
from contextlib import contextmanager

@contextmanager
def retry_on_fail(max_attempts):
    try:
        yield
    except Exception as e:
        print(f"Attempt failed with: {e}")
        for attempt in range(2, max_attempts + 1):
            print(f"Remaining retries: {max_attempts - attempt + 1}. Printing plc
        # Re-raise the exception to exit the context normally after 'mock' retrie
```

```
        raise

try:
    with retry_on_fail(3):
        raise RuntimeError("Network issue.")
except RuntimeError as e:
    print(f"Outer block caught: {e.__class__.__name__}")
```

Expected Output:

```
Attempt failed with: Network issue.
Remaining retries: 2. Printing placeholder for logic.
Remaining retries: 1. Printing placeholder for logic.
Outer block caught: RuntimeError
```

Exercise 6.4: Context Manager for Block Timing with Argument Create a context manager factory `time_block(label)` that accepts a string `label` and prints the duration of the enclosed block using that label.

```
from contextlib import contextmanager
import time

@contextmanager
def time_block(label):
    start_time = time.time()
    try:
        yield
    finally:
        end_time = time.time()
        duration = end_time - start_time
        print(f"[{label}] finished in {duration:.4f} seconds.")

with time_block(???):
    time.sleep(0.01)
```

Expected Behavior: The output should show `[DB_FETCH] finished in 0.01xx seconds.`

Exercise 6.5: Suppressing *All* Errors Create a context manager `silent_fail` that suppresses *any* exception raised inside the `with` block, printing a generic "Error suppressed silently" message.

```
from contextlib import contextmanager

@contextmanager
def silent_fail():
    try:
        yield
    except Exception:
        print("Error suppressed silently.")
```

```
        # No re-raise means the exception is handled/suppressed

with silent_fail():
    print("Pre-crash")
    1 / 0
    print("Post-crash (should not run)")

print("Execution continues outside the block.")
```

Expected Output:

```
Pre-crash
Error suppressed silently.
Execution continues outside the block.
```

Hard

Exercise 6.6: Conditional Suppression CM Factory Create a context manager factory `suppress_unless(error_type_to_propagate)` that suppresses *all* exceptions that occur, *except* for the one specified in the factory argument, which must be re-raised (propagated).

```
from contextlib import contextmanager

@contextmanager
def suppress_unless(error_type_to_propagate):
    try:
        yield
    except error_type_to_propagate:
        print(f"PROPAGATING: {error_type_to_propagate.__name__}")
        raise # Reraise the specified error
    except Exception as e:
        print(f"SUPPRESSED: {e.__class__.__name__} was ignored.")
        # Suppress all others

# Test 1: Suppress (default behavior)
with suppress_unless(ValueError):
    raise IndexError("Ignored")
print("---")
# Test 2: Propagate (ValueError specified)
try:
    with suppress_unless(ValueError):
        raise ValueError("Propagate me")
except ValueError as e:
    print(f"Caught propagated error: {e.__class__.__name__}")
```

Expected Output:

```
SUPPRESSED: IndexError was ignored.
---
PROPAGATING: ValueError
```

```
Caught propagated error: ValueError
```

Exercise 6.7: Context Manager for Input Validation Create a context manager `validate_input(input_value, validation_func)` that checks `input_value` against `validation_func` on entry. If the validation fails (returns `False`), it raises a `ValueError` immediately, preventing the block from running.

```
from contextlib import contextmanager

def is_non_empty_string(value):
    return isinstance(value, str) and len(value) > 0

@contextmanager
def validate_input(input_value, validation_func):
    if not ???(input_value):
        raise ValueError(f"Input '{input_value}' failed validation.")
    try:
        yield
    finally:
        pass # No cleanup needed

try:
    with validate_input("Valid string", is_non_empty_string):
        print("Block 1 runs.")
    with validate_input("", is_non_empty_string):
        print("Block 2 runs. (Should not reach)")
except ValueError as e:
    print(f"Caught validation error: {e}")
```

Expected Output:

```
Block 1 runs.
Caught validation error: Input '' failed validation.
```

Exercise 6.8: Nested Lock Management with Yielded Resource Nest the `safe_lock` context manager (from 5.8) inside a `resource_allocator` context manager that yields a temporary resource ID. Observe the lock management and resource allocation order.

```
from contextlib import contextmanager
import threading

# Assume safe_lock is defined as in 5.8
@contextmanager
def safe_lock(lock):
    print("LOCK: Acquiring...")
    lock.acquire()
    try:
        yield
    finally:
        lock.release()
```

```
        print("LOCK: Released.")

@contextmanager
def resource_allocator(name):
    print(f"RESOURCE: Allocating {name}")
    try:
        yield f"ID_{name}"
    finally:
        print(f"RESOURCE: Deallocating {name}")

my_nested_lock = threading.Lock()

with safe_lock(my_nested_lock):
    with resource_allocator("A") as res_a:
        print(f"Working with {res_a} while locked.")
    print("Lock held during resource allocation/deallocation.")
```

Expected Output:

```
LOCK: Acquiring...
RESOURCE: Allocating A
Working with ID_A while locked.
RESOURCE: Deallocating A
Lock held during resource allocation/deallocation.
LOCK: Released.
```

7️⃣ Mixed Decorators + Context Managers

This section focuses on scenarios where these two powerful tools interact.

Medium

Exercise 7.1: Decorator that Uses a Context Manager Create a decorator `with_timer` that wraps the decorated function call within the `timer_cm` context manager (from 5.3) to automatically time the function's execution.

```
from functools import wraps
from contextlib import contextmanager
import time

@contextmanager
def timer_cm():
    start_time = time.time()
    try:
        yield
    finally:
        end_time = time.time()
        duration = end_time - start_time
        print(f"Function executed in {duration:.4f} seconds.")

def with_timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
```

```
        # The key: using the CM inside the decorator's wrapper
        with timer_cm():
            return func(*args, **kwargs)
    return wrapper

@with_timer
def fetch_data():
    time.sleep(0.03)
    return "Data fetched."

print(fetch_data())
```

Expected Behavior: The output shows the timer message (approx 0.03xx seconds) followed by "Data fetched."

Exercise 7.2: Resource Lock Decorator Factory Create a decorator factory `with_lock(lock)` that accepts a `threading.Lock` instance and ensures the decorated function runs only while holding the lock. Use the `safe_lock` context manager (from 5.8) internally.

```
from functools import wraps
from contextlib import contextmanager
import threading
import time

# Re-define safe_lock for completeness in this section
@contextmanager
def safe_lock(lock):
    lock.acquire()
    try:
        yield
    finally:
        lock.release()

def with_lock(lock):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Use the CM here
            with safe_lock(lock):
                return func(*args, **kwargs)
        return wrapper
    return decorator

SHARED_LOCK = threading.Lock()

@with_lock(SHARED_LOCK)
def update_shared_resource(value):
    print(f"Lock status inside function: {SHARED_LOCK.locked()}")
    time.sleep(0.01)
    return f"Updated with {value}"

print(update_shared_resource(100))
print(f"Lock status outside function: {SHARED_LOCK.locked()}")
```

Expected Output:

```
Lock status inside function: True
Updated with 100
Lock status outside function: False
```

Hard

Exercise 7.3: Caching with Time-Limited Context Create a context manager `temporary_cache` that, on entry, temporarily switches the function `compute_val` to use a global cache (by swapping out the original function with a cached version) and restores the original function on exit.

```
from contextlib import contextmanager

GLOBAL_CACHE_MOCK = {}
GLOBAL_CALC_COUNT = 0

def original_compute_val(x):
    global GLOBAL_CALC_COUNT
    GLOBAL_CALC_COUNT += 1
    return x * 10

def cached_compute_val(x):
    if x not in GLOBAL_CACHE_MOCK:
        GLOBAL_CACHE_MOCK[x] = original_compute_val(x)
    return GLOBAL_CACHE_MOCK[x]

# Function holder (this is what the user calls)
compute_val = original_compute_val

@contextmanager
def temporary_cache():
    global compute_val
    original_func = compute_val
    compute_val = cached_compute_val # Switch to cached version
    try:
        yield
    finally:
        compute_val = original_func # Switch back

# Test 1: No cache
print(f"No Cache: {compute_val(1)} | Count: {GLOBAL_CALC_COUNT}")

# Test 2: Inside cache
with temporary_cache():
    print(f"In Cache: {compute_val(2)} | Count: {GLOBAL_CALC_COUNT}") # Calc 1
    print(f"In Cache: {compute_val(2)} | Count: {GLOBAL_CALC_COUNT}") # Cache hit

# Test 3: Back to no cache
print(f"No Cache: {compute_val(3)} | Count: {GLOBAL_CALC_COUNT}") # Calc 2
```

Expected Output:

```
No Cache: 10 | Count: 1
In Cache: 20 | Count: 2
In Cache: 20 | Count: 2
No Cache: 30 | Count: 3
```

Exercise 7.4: Decorator Factory that Implements State Management Create a decorator factory `with_state_prefix(prefix)` that wraps the decorated function's call in a context manager. This internal context manager should temporarily set a `CURRENT_PREFIX` global variable to the provided `prefix` and restore it afterward.

```
from functools import wraps
from contextlib import contextmanager

CURRENT_PREFIX = ""

@contextmanager
def state_prefix_manager(prefix):
    global CURRENT_PREFIX
    original_prefix = CURRENT_PREFIX
    CURRENT_PREFIX = prefix
    try:
        yield
    finally:
        CURRENT_PREFIX = original_prefix

def with_state_prefix(prefix):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Use CM inside decorator
            with state_prefix_manager(prefix):
                return func(*args, **kwargs)
        return wrapper
    return decorator

def message(text):
    print(f"[{CURRENT_PREFIX}] {text}")

message("Initial message.")

@with_state_prefix("API")
def api_call(endpoint):
    message(f"Calling {endpoint}...")

@with_state_prefix("DB")
def db_query(sql):
    message(f"Executing {sql}...")

api_call("/users")
db_query("SELECT *")
message("Final message.")
```

Expected Output:

```
[] Initial message.
[API] Calling /users...
[DB] Executing SELECT *...
[] Final message.
```

Exercise 7.5: Function-based Context Manager that Returns a Decorator Create a context manager `decorate_block(decorator)` that, on entry, applies the given decorator to a *mock* function (a closure or nested function), yields the decorated function, and then restores the original mock function on exit.

(This is purely conceptual to show the return value of `__enter__` / `yield` can be anything, even a dynamically generated decorator.)

```
from contextlib import contextmanager
from functools import wraps

# Simple logging decorator for demonstration
def simple_log(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("LOGGED EXECUTION.")
        return func(*args, **kwargs)
    return wrapper

@contextmanager
def decorate_block(decorator):
    print("CM: Applying decorator...")
    def mock_function():
        return "Original Call."

    # Decorate the mock function
    decorated_func = decorator(mock_function)

    try:
        yield decorated_func
    finally:
        print("CM: Context exited.")

# Usage: Capture the decorated function as 'action'
with decorate_block(simple_log) as action:
    result = action()
    print(result)
```

Expected Output:

```
CM: Applying decorator...
LOGGED EXECUTION.
Original Call.
```

```
CM: Context exited.
```

8 Paragraphic / Scenario-based Hard Exercises

These exercises require you to analyze the requirements and write the complete, correct, and robust code solution without any starter template.

Exercise 8.1: API Call Timer and Logger You need a system for monitoring external API calls. Design a single decorator factory, `monitor_api_call(service_name)`, that takes the name of the API service (e.g., "UserAuth", "ProductDB"). This decorator must: 1) Log the function name and service name before execution. 2) Log the time taken (in seconds) for the function to complete. 3) Log the returned value. 4) Use `functools.wraps`.

Requirements: Decorator factory, logging, timing, `functools.wraps`. **Expected Outcome:** Calling a decorated function `fetch_user(id)` should print three lines of logs (CALL, TIME, RETURN) before printing the function's result.

Exercise 8.2: Database Transaction Context Manager (Mock) Create a generator-based context manager, `db_transaction(db_conn)`, that simulates a database transaction. It must: 1) Print "Transaction BEGIN" on entry. 2) If the block executes without error, print "Transaction COMMIT". 3) If *any* exception occurs, print "Transaction ROLLBACK" and **re-raise** the original exception so the calling code is aware of the failure.

Requirements: Generator-based CM, exception handling, propagation. **Expected Outcome:** A `ZeroDivisionError` raised inside the `with` block should trigger "Transaction ROLLBACK" and then be caught by an external `try...except` block.

Exercise 8.3: Access Control with Default Role Factory Design a decorator factory, `restrict_to(required_role)`, that checks a function against a global `USER_PERMISSIONS` dictionary (where the key is the function name and the value is the current user's role, e.g., `{'delete_data': 'guest'}`). If the user's current role does not match the `required_role`, raise a `PermissionError` including the function name and the required role. If the role matches, execute the function.

Requirements: Decorator factory, access to global state, specific exception raising. **Expected Outcome:** A function decorated with `@restrict_to('admin')` called when `USER_PERMISSIONS` shows 'guest' for that function should raise `PermissionError`.

Exercise 8.4: Read-Only State Context Manager Create a generator-based context manager, `read_only_mode`, that sets a global boolean `CAN_WRITE` to `False` on entry, and always restores it to `True` on exit. If the context manager is entered while `CAN_WRITE` is already `False` (nested/re-entry), it should silently proceed but the outer manager must be responsible for restoring the state.

Requirements: Generator-based CM, handling global boolean state, ensuring restore on exit, handling nested state. **Expected Outcome:** `CAN_WRITE` should be `False` inside the block, and

`True` outside, even if an exception occurs inside.

Exercise 8.5: Combined Validation and Logging Decorator Create a decorator, `log_and_validate`, that performs two actions sequentially: 1) It ensures that the first positional argument is a non-negative number (raises `ValueError` otherwise). 2) If successful, it logs the function name and the result *after* execution.

Requirements: Single decorator, input validation, output logging, error propagation. **Expected Outcome:** Calling the decorated function with `-5` should immediately raise `ValueError` and the function should not execute.

Exercise 8.6: Custom Error Mapping Context Manager (Factory) Design a context manager factory, `map_error(original_error, target_error)`, that accepts two exception types. It should catch the `original_error` if raised within the `with` block and re-raise it as the `target_error`.

Requirements: Generator-based CM factory, exception translation/conversion. **Expected Outcome:** Using `@map_error(ZeroDivisionError, ValueError)` and raising `1/0` inside the block should result in a `ValueError` being caught outside.

Exercise 8.7: Decorator for Argument Type Checking (Multiple) Create a decorator, `check_types(int_args=[], str_args=[])`, that accepts lists of argument names that must be integers and strings, respectively. It should check the types of these arguments (passed as kwargs only, for simplicity) and raise a `TypeError` if any type mismatch is found.

Requirements: Decorator factory, checking multiple argument types via kwargs, custom `TypeError` message. **Expected Outcome:** A function decorated with `@check_types(int_args=['age'], str_args=['name'])` called with `age="25"` should raise `TypeError`.

Exercise 8.8: Dynamic Resource Yielding Context Manager Create a context manager, `dynamic_resource_cm(config_name)`, that simulates resource allocation. It should yield a dictionary `{ 'status': 'READY', 'config': config_name }`. On exit, it should print the status and ensure the yielded dictionary's status is updated to 'CLOSED' (demonstrating that the yielded object is mutable).

Requirements: Generator-based CM, yielding a mutable object, updating the yielded object on cleanup. **Expected Outcome:** The variable captured by the `as` clause will have its internal status changed from 'READY' to 'CLOSED' after the `with` block finishes.

Exercise 8.9: Sequential Decorator and Context Manager Interaction Implement the following scenario: A function `calculate(a, b)` is decorated with `@time_it` (from 2.8). The entire function call is then wrapped in a context manager, `suppress_io_errors`, that suppresses `IOError` but propagates all others. Raise a `TimeoutError` inside the function and observe the execution flow.

Requirements: Decorator (`time_it`), CM (`suppress_io_errors`), exception flow observation. **Expected Outcome:** The `time_it` logs START, the CM setup runs, the function executes, the

`TimeoutError` bypasses the CM suppression, the CM teardown runs, the `time_it` logs TIME/END, and the `TimeoutError` is caught by an external handler.

Exercise 8.10: Conditional Caching Decorator Factory Design a decorator factory, `cache_if(condition_arg_name)`, that accepts the name of a boolean keyword argument. If this argument is `True` when the function is called, the result is cached; otherwise, the function executes normally, and the result is not stored in the cache.

Requirements: Decorator factory, conditional caching based on kwarg, argument-based cache key. **Expected Outcome:** A function decorated with `@cache_if('use_cache')` called with `use_cache=True` caches the result. A subsequent call with `use_cache=False` executes the function again, regardless of the cache contents.

Exercise 8.11: Nested Lock with Timeout Context Manager (Mock) Create a context manager factory, `resource_lock_with_timeout(lock, timeout)`, that attempts to acquire the provided `threading.Lock`. If the lock acquisition fails (simulated by checking if the lock is already acquired by another function), it should raise a `TimeoutError`. The actual lock acquisition and release must occur, ensuring the lock is always released.

Requirements: CM factory, `threading.Lock` usage, handling lock status, exception raising. **Expected Outcome:** A second simultaneous call to the same lock (using mock state or a non-blocking check) should raise `TimeoutError`, but the original lock must remain in a consistent state (released).

Exercise 8.12: Debugger Mode Context Manager Create a context manager `debug_mode` that temporarily sets a global boolean `IS_DEBUG` to `True` on entry. Additionally, it must yield a boolean value (True) that can be captured by the `as` clause. On exit, it restores `IS_DEBUG` to `False`. Use this manager to conditionally print debug messages inside the block.

Requirements: Generator-based CM, handling global state, yielding a value, conditional logic inside the block. **Expected Outcome:** The `IS_DEBUG` state changes correctly, and the code inside the block uses the yielded value to control debug output.

Exercise 8.13: Multi-Stage Validation Decorator Create a decorator, `multi_stage_validation(arg_name, check_funcs)`, that accepts an argument name and a list of single-argument validation functions. The function should be executed only if all validation functions return `True` for the specified argument. If any fails, raise a `ValueError` indicating which check failed.

Requirements: Decorator factory, list of functions as arguments, sequential checks. **Expected Outcome:** A function decorated with `@multi_stage_validation('value', [check1, check2])` should run only if both `check1(value)` and `check2(value)` are True.

Exercise 8.14: Environment Variable Context Manager Create a context manager factory, `temp_environment(key, value)`, that temporarily sets a global dictionary `ENV_VARS` key to a new value on entry and restores the original value (or deletes the key if it didn't exist) on exit.

Requirements: CM factory, handling dictionary state, restoring original value or cleanup.
Expected Outcome: The global `ENV_VARS` dictionary is modified inside the block and reverted to its initial state outside the block.

Exercise 8.15: Decorator that Enforces Context Manager Usage Design a decorator, `requires_context_manager`, that is designed to wrap a function that takes a connection object as its first argument. The decorator should check whether the passed argument is an active resource (e.g., checks if it has a specific attribute like `is_active=True`). If the resource is not active (i.e., it wasn't opened by a context manager), the decorator should raise a `RuntimeError`.

(Note: This is a conceptual check; the check should confirm the argument object has the required state set by a CM.)

Requirements: Decorator, checking state of an argument, raising `RuntimeError`. **Expected Outcome:** Calling `process(conn)` should raise `RuntimeError` if `conn.is_active` is False, and execute successfully if it is True.

9️⃣ Solutions

This section provides complete, runnable solutions for all exercises, along with detailed explanations and conceptual callouts.

Decorators – Basics Solutions

Exercise 1.1 Solution

```
def add_greeting(func):
    def wrapper():
        print("Hello!")
        func() # Execute the decorated function
    return wrapper

@add_greeting
def say_name():
    print("My name is Alex.")

say_name()
# Output:
# Hello!
# My name is Alex.
```

Exercise 1.2 Solution

```
def check_alias(func):
    def wrapper():
        print(f"Executing function: {func.__name__}")
        func()
    return wrapper

@check_alias
```

```
def calculate_sum():
    print(10 + 20)

calculate_sum()
# Output:
# Executing function: calculate_sum
# 30
```

Exercise 1.3 Solution

```
def post_message(func):
    def wrapper():
        func() # Execute the decorated function first
        print("Execution complete.")
    return wrapper

@post_message
def cleanup():
    print("Cleaning up resources.")

cleanup()
# Output:
# Cleaning up resources.
# Execution complete.
```

Exercise 1.4 Solution

```
def accept_everything(func):
    def wrapper(*args, **kwargs):
        print(f"Args: {args}")
        print(f"Kwargs: {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@accept_everything
def describe_item(name, price, stock=True):
    return f"Item: {name}, Price: ${price}, In Stock: {stock}"

result = describe_item("Laptop", 1200, stock=False)
print(result)
# Output:
# Args: ('Laptop', 1200)
# Kwargs: {'stock': False}
# Item: Laptop, Price: $1200, In Stock: False

# Callout: The use of *args and **kwargs in the wrapper signature ensures the dec
```

Exercise 1.5 Solution

```
def double_result(func):
    def wrapper(*args, **kwargs):
```

```
        original_result = func(*args, **kwargs)
        return original_result * 2
    return wrapper

@double_result
def get_number(value):
    return value

print(get_number(10))
# Output:
# 20
```

Exercise 1.6 Solution

```
from functools import wraps

def debug_no_wraps(func):
    def wrapper(*args, **kwargs):
        print("DEBUG: Running function.")
        return func(*args, **kwargs)
    return wrapper

def debug_with_wraps(func):
    @wraps(func) # Use functools.wraps here
    def wrapper(*args, **kwargs):
        print("DEBUG: Running function.")
        return func(*args, **kwargs)
    return wrapper

@debug_no_wraps
def compute_data_a():
    """Calculates data A."""
    return 1

@debug_with_wraps
def compute_data_b():
    """Calculates data B."""
    return 2

# Check metadata
print(f"A Name (No Wraps): {compute_data_a.__name__}")
print(f"B Name (With Wraps): {compute_data_b.__name__}")
print(f"A Doc (No Wraps): {compute_data_a.__doc__}")
print(f"B Doc (With Wraps): {compute_data_b.__doc__}")
# Output:
# A Name (No Wraps): wrapper
# B Name (With Wraps): compute_data_b
# A Doc (No Wraps): None
# B Doc (With Wraps): Calculates data B.

# Callout: functools.wraps is essential. Without it, the decorated function loses
```

Exercise 1.7 Solution

```
from functools import wraps

def run_if_positive(func):
    @wraps(func)
    def wrapper(number, *args, **kwargs):
        if number > 0:
            return func(number, *args, **kwargs)
        else:
            print(f"ERROR: Cannot run {func.__name__} with non-positive input: {n
            return None # Explicitly return None
    return wrapper

@run_if_positive
def safe_divide(x, y):
    return x / y

print(safe_divide(5, 2))
print(safe_divide(-1, 5))
# Output:
# 2.5
# ERROR: Cannot run safe_divide with non-positive input: -1
# None
```

Decorators – Advanced Patterns Solutions

Exercise 2.1 Solution

```
from functools import wraps

def log_call(func):
    @wraps(func)
    def wrapper(a, b):
        # Use func.__name__ and the arguments passed to the wrapper
        print(f"LOG: Calling {func.__name__} with args: ({a}, {b})")
        return func(a, b)
    return wrapper

@log_call
def multiply(x, y):
    return x * y

print(multiply(8, 7))
# Output:
# LOG: Calling multiply with args: (8, 7)
# 56
```

Exercise 2.2 Solution

```
from functools import wraps

def timing_start_end(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
```

```
        print(f"--- START: {func.__name__}")
        result = func(*args, **kwargs) # Execute and capture result
        print(f"--- END: {func.__name__}")
        return result
    return wrapper

@timing_start_end
def process_data():
    return [x * 2 for x in range(3)]

print(process_data())
# Output:
# --- START: process_data
# --- END: process_data
# [0, 2, 4]
```

Exercise 2.3 Solution

```
from functools import wraps

def validate_integer(func):
    @wraps(func)
    def wrapper(value, *args, **kwargs):
        if not isinstance(value, int):
            raise TypeError(f"Input must be an integer, got {type(value).__name__
        return func(value, *args, **kwargs)
    return wrapper

@validate_integer
def process_id(user_id):
    return f"Processing ID: {user_id}"

# Test cases
print(process_id(123))
try:
    process_id("abc") # Raises TypeError
except TypeError as e:
    print(e)
# Output:
# Processing ID: 123
# Input must be an integer, got str
```

Exercise 2.4 Solution

```
from functools import wraps

def log_io(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"IO LOG: {func.__name__} called with args: {args}, kwargs: {kwargs
        result = func(*args, **kwargs)
        print(f"IO LOG: {func.__name__} returned: {result}") # Log the captured r
        return result
    return wrapper
```

```
@log_io
def power(base, exp=2):
    return base ** exp

print(power(3, exp=3))
# Output:
# IO LOG: power called with args: (3,), kwargs: {'exp': 3}
# IO LOG: power returned: 27
# 27
```

Exercise 2.5 Solution

```
from functools import wraps

CACHE = {}

def simple_cache(func):
    @wraps(func)
    def wrapper():
        if func.__name__ in CACHE:
            print("From cache.")
            return CACHE[func.__name__]
        else:
            result = func()
            CACHE[func.__name__] = result
            return result
    return wrapper

@simple_cache
def expensive_call():
    print("Performing expensive calculation...")
    return 42

print(expensive_call())
print(expensive_call())
# Output:
# Performing expensive calculation...
# 42
# From cache.
# 42
```

Exercise 2.6 Solution

```
from functools import wraps

CURRENT_USER_ROLE = "viewer"

def requires_admin(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        global CURRENT_USER_ROLE
        if CURRENT_USER_ROLE == "admin":
            return func(*args, **kwargs)
```

```
        else:
            raise PermissionError(f"User '{CURRENT_USER_ROLE}' is not authorized
    return wrapper

@requires_admin
def delete_database():
    return "Database deleted successfully."

try:
    delete_database()
except PermissionError as e:
    print(e)

CURRENT_USER_ROLE = "admin"
print(delete_database())
# Output:
# User 'viewer' is not authorized to run 'delete_database'
# Database deleted successfully.
```

Exercise 2.7 Solution

```
from functools import wraps

ADVANCED_CACHE = {}

def smart_cache(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Normalize kwargs for a stable key: sort items and convert to tuple
        key_parts = [func.__name__, args, tuple(sorted(kwargs.items()))]
        cache_key = tuple(key_parts)

        if cache_key in ADVANCED_CACHE:
            print("From advanced cache.")
            return ADVANCED_CACHE[cache_key]
        else:
            result = func(*args, **kwargs)
            ADVANCED_CACHE[cache_key] = result
            return result
    return wrapper

@smart_cache
def fibonacci(n):
    print(f"Calculating fibonacci({n})...")
    if n <= 1: return n
    # Note: Recursive calls here will also be cached, but only for the same signa
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(4)) # Calculates fully, stores all intermediate steps
print(fibonacci(4)) # Cache hit for 4
print(fibonacci(3)) # Cache hit for 3 (stored during first call to fib(4))
# Output:
# Calculating fibonacci(4)...
# Calculating fibonacci(3)...
# Calculating fibonacci(2)...
# Calculating fibonacci(1)...
# Calculating fibonacci(0)...
```

```
# From advanced cache.
# From advanced cache.
# 1
# 3
# From advanced cache.
# 3
# From advanced cache.
# 2
```

Exercise 2.8 Solution

```
from functools import wraps
import time

def time_it(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        duration = end_time - start_time
        print(f"Time: {func.__name__} took {duration:.4f} seconds.")
        return result
    return wrapper

@time_it
def delay_task(seconds):
    time.sleep(seconds)
    return f"Slept for {seconds}s"

print(delay_task(0.05))
# Output (approx):
# Time: delay_task took 0.0500 seconds.
# Slept for 0.05s
```

Decorators with Arguments & Stacking Solutions

Exercise 3.1 Solution

```
from functools import wraps

def add_prefix(prefix): # Decorator Factory
    def decorator(func): # Actual Decorator
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(f"[{prefix}]")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@add_prefix("STATUS")
def report_status():
    print("System is nominal.")
```

```
report_status()
# Output:
# [STATUS]
# System is nominal.

# Callout: The closure captures the 'prefix' argument from the outer factory func
```

Exercise 3.2 Solution

```
from functools import wraps

def run_n_times(n):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for i in range(n):
                print(f"Run {i+1}/{n}: ", end="")
                func(*args, **kwargs)
        return wrapper
    return decorator

@run_n_times(3)
def blast_off():
    print("T-minus 10...")

blast_off()
# Output:
# Run 1/3: T-minus 10...
# Run 2/3: T-minus 10...
# Run 3/3: T-minus 10...
```

Exercise 3.3 Solution

```
def add_greeting(func):
    def wrapper():
        print("Hello!")
        func()
    return wrapper

def post_message(func):
    def wrapper():
        func()
        print("Execution complete.")
    return wrapper

@post_message # Outer decorator (executed last)
@add_greeting # Inner decorator (executed first, wraps original function)
def perform_task():
    print("Performing task...")

perform_task()
# Execution Order Trace:
# 1. perform_task is passed to @add_greeting -> becomes add_greeting.wrapper
# 2. add_greeting.wrapper is passed to @post_message -> becomes post_message.wrap
```

```
# 3. Call perform_task() -> calls post_message.wrapper
# 4. post_message.wrapper calls its wrapped function (add_greeting.wrapper)
# 5. add_greeting.wrapper prints "Hello!"
# 6. add_greeting.wrapper calls its wrapped function (perform_task)
# 7. perform_task prints "Performing task..."
# 8. add_greeting.wrapper returns
# 9. post_message.wrapper prints "Execution complete."
# Output:
# Hello!
# Performing task...
# Execution complete.
```

Exercise 3.4 Solution

```
from functools import wraps

LOG_ORDER = {"DEBUG": 1, "INFO": 2, "WARN": 3, "ERROR": 4}
LOG_THRESHOLD = "INFO" # Only INFO (2) and ERROR (4) will run

def log_with_level(level):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check if the decorator's level is >= the global threshold
            if LOG_ORDER.get(level, 5) >= LOG_ORDER.get(LOG_THRESHOLD, 5):
                print(f"[{level}] {func.__name__} called.")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@log_with_level("INFO")
def status_check():
    return "Status OK"

@log_with_level("DEBUG")
def detailed_log():
    return "Detailed data"

print(status_check())
print(detailed_log())
# Output:
# [INFO] status_check called.
# Status OK
# Detailed data
```

Exercise 3.5 Solution

```
from functools import wraps

def validate_type(arg_name, expected_type):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check kwargs first
```

```
            if arg_name in kwargs and not isinstance(kwargs[arg_name], expected_t
                raise TypeError(f"Argument '{arg_name}' must be of type {expected
            # Simplified check for positional arguments (assuming arg_name is the
            return func(*args, **kwargs)
        return wrapper
    return decorator

@validate_type(arg_name="limit", expected_type=int)
def fetch_records(query, limit=10):
    return f"Fetching '{query}' with limit {limit}"

print(fetch_records("users", limit=50))
try:
    fetch_records("products", limit="all")
except TypeError as e:
    print(e)
# Output:
# Fetching 'users' with limit 50
# Argument 'limit' must be of type int, got str
```

Exercise 3.6 Solution

```
from functools import wraps

def add_prefix(prefix):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(f"[{prefix}]", end=" ")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@add_prefix("OUTER") # 1. Applied second (Outer Wrapper)
@add_prefix("INNER") # 2. Applied first (Inner Wrapper)
def process_data():
    print("Data processed.")

process_data()
# Execution Order Trace:
# The call goes to the OUTER wrapper, which prints its prefix, and then calls the
# The INNER wrapper prints its prefix, and then calls the original function.
# Output:
# [OUTER] [INNER] Data processed.
```

Exercise 3.7 Solution

```
from functools import wraps
import time

attempt_count = 0 # Global counter for demonstration

def retry(max_attempts, delay=0):
    def decorator(func):
```

```
        @wraps(func)
        def wrapper(*args, **kwargs):
            global attempt_count
            for attempt in range(1, max_attempts + 1):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_attempts:
                        print(f"ERROR: Max retries ({max_attempts}) reached. Rerc
                        raise
                    print(f"WARNING: Exception '{e}' occurred. Retrying in {delay
                    time.sleep(delay)
                    attempt_count += 1
        return wrapper
    return decorator

@retry(max_attempts=3, delay=0.01)
def unstable_network_call():
    global attempt_count
    if attempt_count < 2: # Fail on the first two runs (attempt_count 0 and 1)
        raise IOError("Connection lost.")
    return "Success after retry."

print(unstable_network_call())
# Output:
# WARNING: Exception 'Connection lost.' occurred. Retrying in 0.01s (Attempt 1/3)
# WARNING: Exception 'Connection lost.' occurred. Retrying in 0.01s (Attempt 2/3)
# Success after retry.
```

Exercise 3.8 Solution

```
from functools import wraps

GLOBAL_CACHE = {}
KEY_ORDER = [] # Tracks insertion order for LIFO eviction

def cache_for_args(cache_size):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            key_parts = [func.__name__, args, tuple(sorted(kwargs.items()))]
            cache_key = tuple(key_parts)

            if cache_key in GLOBAL_CACHE:
                return GLOBAL_CACHE[cache_key]

            # Cache miss - perform computation
            result = func(*args, **kwargs)

            # Check for eviction
            if len(GLOBAL_CACHE) >= cache_size:
                oldest_key = KEY_ORDER.pop(0) # Remove oldest key
                del GLOBAL_CACHE[oldest_key]
                print(f"CACHE: Evicting {oldest_key[0]} to make space.")

            # Store new result
            GLOBAL_CACHE[cache_key] = result
```

```
            KEY_ORDER.append(cache_key)
            return result
        return wrapper
    return decorator

@cache_for_args(cache_size=2)
def add(a, b):
    print(f"Calculating {a} + {b}...")
    return a + b

print(add(1, 1)) # Key 1: calculated (Order: [1])
print(add(2, 2)) # Key 2: calculated (Order: [1, 2])
print(add(3, 3)) # Key 3: calculated (Evicts Key 1. Order: [2, 3])
print(add(1, 1)) # Key 1: Should calculate again after eviction
# Output:
# Calculating 1 + 1...
# 2
# Calculating 2 + 2...
# 4
# Calculating 3 + 3...
# CACHE: Evicting add to make space.
# 6
# Calculating 1 + 1...
# 2
```

Exercise 3.9 Solution

```
from functools import wraps

# Decorator 1: Integer Check
def validate_int(func):
    @wraps(func)
    def wrapper(value):
        if not isinstance(value, int):
            raise TypeError("Value must be an integer.")
        return func(value)
    return wrapper

# Decorator Factory 1: Range Check
def validate_range(min_val, max_val):
    def decorator(func):
        @wraps(func)
        def wrapper(value):
            if not (min_val <= value <= max_val):
                raise ValueError(f"Value must be between {min_val} and {max_val}.
            return func(value)
        return wrapper
    return decorator

# Decorator 2: Positive Check
def validate_positive(func):
    @wraps(func)
    def wrapper(value):
        if value <= 0:
            raise ValueError("Value must be positive.")
        return func(value)
    return wrapper
```

```
@validate_range(10, 20) # 3. Outer
@validate_positive # 2. Middle
@validate_int # 1. Inner
def process_score(score):
    return f"Score {score} is valid."

print(process_score(15))

try:
    process_score(5)
except ValueError as e:
    print(f"Caught Error: {e}") # Should be caught by validate_positive (Middle)

try:
    process_score(25)
except ValueError as e:
    print(f"Caught Error: {e}") # Should be caught by validate_range (Outer)

try:
    process_score(15.5)
except TypeError as e:
    print(f"Caught Error: {e}") # Should be caught by validate_int (Inner)

# Output:
# Score 15 is valid.
# Caught Error: Value must be positive.
# Caught Error: Value must be between 10 and 20.
# Caught Error: Value must be an integer.
```

Context Managers – Basics Solutions

Exercise 4.1 Solution

```
from contextlib import contextmanager

@contextmanager
def greet_context():
    print("Entering block") # Setup
    try:
        yield # Execution point
    finally:
        print("Exiting block") # Teardown

with greet_context():
    print("Inside the block.")
# Output:
# Entering block
# Inside the block.
# Exiting block

# Callout: In a generator-based CM, the code before 'yield' is __enter__ logic, c
```

Exercise 4.2 Solution

```
from contextlib import contextmanager

@contextmanager
def mock_connection(name):
    print(f"Connecting to {name}...")
    try:
        yield # Yield nothing, just a pause
    finally:
        print(f"Closing connection to {name}.")

with mock_connection("DB_A"):
    print("Performing queries...")
# Output:
# Connecting to DB_A...
# Performing queries...
# Closing connection to DB_A.
```

Exercise 4.3 Solution

```
from contextlib import contextmanager

GLOBAL_SETTING = "DEFAULT"

@contextmanager
def temp_state(variable_name, temp_value):
    global GLOBAL_SETTING
    original_value = GLOBAL_SETTING
    print(f"Old setting: {original_value}")
    GLOBAL_SETTING = temp_value
    try:
        yield
    finally:
        GLOBAL_SETTING = original_value
        print(f"Setting restored: {GLOBAL_SETTING}")

with temp_state("GLOBAL_SETTING", "TEST_MODE"):
    print(f"Inside with: {GLOBAL_SETTING}")

print(f"Outside with: {GLOBAL_SETTING}")
# Output:
# Old setting: DEFAULT
# Inside with: TEST_MODE
# Setting restored: DEFAULT
# Outside with: DEFAULT
```

Exercise 4.4 Solution

```
from contextlib import contextmanager
import threading

my_lock = threading.Lock()

@contextmanager
def lock_manager(lock):
```

```
    print("Attempting to acquire lock...")
    lock.acquire()
    try:
        yield
    finally:
        lock.release()
        print("Lock released.")

with lock_manager(my_lock):
    print("Lock acquired, critical section running.")
    assert my_lock.locked() == True

assert my_lock.locked() == False
# Output:
# Attempting to acquire lock...
# Lock acquired, critical section running.
# Lock released.
```

Exercise 4.5 Solution

```
from contextlib import contextmanager

@contextmanager
def mock_connection(name):
    print(f"Connecting to {name}...")
    try:
        yield name # Yield the connection object/resource
    finally:
        print(f"Closing connection to {name}.")

with mock_connection("DB_B") as conn:
    print(f"Using connection: {conn}")
    assert conn == "DB_B"
# Output:
# Connecting to DB_B...
# Using connection: DB_B
# Closing connection to DB_B.
```

Exercise 4.6 Solution

```
from contextlib import contextmanager

@contextmanager
def greet_context(tag):
    print(f"Entering block: {tag}")
    try:
        yield
    finally:
        print(f"Exiting block: {tag}")

with greet_context("Outer"):
    print("Outer action.")
    with greet_context("Inner"):
        print("Inner action.")
    print("Back in Outer.")
```

```
# Output:
# Entering block: Outer
# Outer action.
# Entering block: Inner
# Inner action.
# Exiting block: Inner
# Back in Outer.
# Exiting block: Outer

# Callout: Nested CMs execute their __enter__ (setup) logic from outer to inner,
```

Exercise 4.7 Solution

```
from contextlib import contextmanager

GLOBAL_COUNT = 0

@contextmanager
def usage_tracker(name):
    global GLOBAL_COUNT
    GLOBAL_COUNT += 1
    print(f"[{name}] Entered. Count: {GLOBAL_COUNT}")
    try:
        yield
    finally:
        GLOBAL_COUNT -= 1
        print(f"[{name}] Exited. Count: {GLOBAL_COUNT}")

with usage_tracker("A"):
    with usage_tracker("B"):
        pass

assert GLOBAL_COUNT == 0
# Output:
# [A] Entered. Count: 1
# [B] Entered. Count: 2
# [B] Exited. Count: 1
# [A] Exited. Count: 0
```

Context Managers – Advanced & Exception Handling Solutions

Exercise 5.1 Solution

```
from contextlib import contextmanager

@contextmanager
def exception_observer():
    print("Observer: Setup.")
    try:
        yield
    finally:
        print("Observer: Teardown.")

try:
```

```
    with exception_observer():
        print("Raising error.")
        raise ValueError("Simulated failure.")
except ValueError as e:
    print(f"Caught exception outside: {e}")
# Output:
# Observer: Setup.
# Raising error.
# Observer: Teardown.
# Caught exception outside: Simulated failure.

# Callout: The generator CM's 'finally' block (teardown) is guaranteed to run, ev
```

Exercise 5.2 Solution

```
from contextlib import contextmanager

@contextmanager
def error_suppressor(error_type):
    print("Suppressor active.")
    try:
        yield
    except error_type as e:
        print(f"SUPPRESSED: {e.__class__.__name__} was ignored.")
    except Exception as e:
        raise e
    finally:
        print("Suppressor deactivated.")

with error_suppressor(ZeroDivisionError):
    print(10 / 0) # This should be suppressed

try:
    with error_suppressor(ZeroDivisionError):
        print("No error yet.")
        raise IndexError("Should not be suppressed.")
except IndexError as e:
    print(f"Caught non-suppressed error: {e}")
# Output:
# Suppressor active.
# SUPPRESSED: ZeroDivisionError was ignored.
# Suppressor deactivated.
# Suppressor active.
# No error yet.
# Suppressor deactivated.
# Caught non-suppressed error: Should not be suppressed.
```

Exercise 5.3 Solution

```
from contextlib import contextmanager
import time

@contextmanager
def timer_cm():
```

```
    start_time = time.time()
    try:
        yield
    finally:
        end_time = time.time()
        duration = end_time - start_time
        print(f"Block executed in {duration:.4f} seconds.")

with timer_cm():
    time.sleep(0.02)
    print("Operation done.")
# Output (approx):
# Operation done.
# Block executed in 0.02xx seconds.
```

Exercise 5.4 Solution

```
from contextlib import contextmanager

@contextmanager
def convert_error():
    try:
        yield
    except ValueError as e:
        # Raise a different exception, effectively converting it
        raise TypeError(f"Value error converted: {e}")

try:
    with convert_error():
        print("Converting...")
        raise ValueError("Bad input value.")
except TypeError as e:
    print(f"Successfully caught converted error: {e}")
except ValueError as e:
    print("Error: ValueError was not converted.")
# Output:
# Converting...
# Successfully caught converted error: Value error converted: Bad input value.
```

Exercise 5.5 Solution

```
from contextlib import contextmanager

@contextmanager
def always_cleanup():
    print("Resource allocated.")
    try:
        yield
    finally:
        print("Resource always de-allocated.")

try:
    with always_cleanup():
        print("Working...")
        1 / 0
```

```
except ZeroDivisionError:
    print("Exception handled outside.")
# Output:
# Resource allocated.
# Working...
# Resource always de-allocated.
# Exception handled outside.
```

Exercise 5.6 Solution

```
from contextlib import contextmanager

@contextmanager
def convert_error_to_key():
    try:
        yield
    except ValueError as e:
        raise KeyError(f"Conversion: {e}")

@contextmanager
def suppress_index_error():
    try:
        yield
    except IndexError:
        print("INDEX ERROR SUPPRESSED.")

# Test 1: IndexError (Inner CM suppresses)
with convert_error_to_key():
    with suppress_index_error():
        try:
            print("Test 1: Raising IndexError...")
            raise IndexError("Oops")
        except:
            pass

# Test 2: ValueError (Inner CM ignores, Outer CM converts)
try:
    with convert_error_to_key():
        with suppress_index_error():
            print("Test 2: Raising ValueError...")
            raise ValueError("Bad value")
except KeyError as e:
    print(f"Test 2: Caught converted error: {e.__class__.__name__}")

# Test 3: TypeError (Neither CM handles)
try:
    with convert_error_to_key():
        with suppress_index_error():
            print("Test 3: Raising TypeError...")
            raise TypeError("Wrong type")
except TypeError as e:
    print(f"Test 3: Caught propagated error: {e.__class__.__name__}")
# Output:
# Test 1: Raising IndexError...
# INDEX ERROR SUPPRESSED.
# Test 2: Raising ValueError...
# Test 2: Caught converted error: KeyError
# Test 3: Raising TypeError...
```

```
# Test 3: Caught propagated error: TypeError

# Callout: In nested context managers, exception handling starts at the innermost
```

Exercise 5.7 Solution

```
from contextlib import contextmanager

RESOURCE_POOL = []

@contextmanager
def pool_manager(resource_name, limit):
    global RESOURCE_POOL
    if len(RESOURCE_POOL) >= limit:
        raise Exception(f"Resource pool full. Cannot allocate {resource_name}.")

    RESOURCE_POOL.append(resource_name)
    print(f"Allocated {resource_name}. Pool size: {len(RESOURCE_POOL)}")
    try:
        yield resource_name
    finally:
        if resource_name in RESOURCE_POOL:
            RESOURCE_POOL.remove(resource_name)
            print(f"De-allocated {resource_name}. Pool size: {len(RESOURCE_POOL)}

# Test 1: Successful allocation
with pool_manager("Connection 1", 2):
    pass

# Test 2: Allocation and full check
try:
    with pool_manager("C2", 1): # Limit is 1
        with pool_manager("C3", 1): # Fails here
            pass
except Exception as e:
    print(f"Caught error: {e}")
# Output:
# Allocated Connection 1. Pool size: 1
# De-allocated Connection 1. Pool size: 0
# Allocated C2. Pool size: 1
# Caught error: Resource pool full. Cannot allocate C3.
# De-allocated C2. Pool size: 0
```

Exercise 5.8 Solution

```
from contextlib import contextmanager
import threading

my_thread_lock = threading.Lock()

@contextmanager
def safe_lock(lock):
    print("Lock acquiring...")
    lock.acquire()
```

```
    try:
        yield
    except Exception as e:
        # If an exception occurs, the 'finally' block is still guaranteed to run,
        # but the exception will be re-raised AFTER the lock is released.
        raise e
    finally:
        # Crucial: Ensure lock.release() runs here
        lock.release()
        print("Lock released.")

try:
    with safe_lock(my_thread_lock):
        print("Critical operation started.")
        1 / 0 # Force an exception
except ZeroDivisionError:
    print("ZeroDivisionError caught outside.")

# Check lock state
print(f"Lock still held? {my_thread_lock.locked()}")
# Output:
# Lock acquiring...
# Critical operation started.
# Lock released.
# ZeroDivisionError caught outside.
# Lock still held? False
```

Generator-based Context Managers Solutions

Exercise 6.1 Solution

```
from contextlib import contextmanager

INDENT_PREFIX = ""

@contextmanager
def indent_print():
    global INDENT_PREFIX
    original_prefix = INDENT_PREFIX
    INDENT_PREFIX = INDENT_PREFIX + "  "
    try:
        yield
    finally:
        INDENT_PREFIX = original_prefix

def custom_print(message):
    print(f"{INDENT_PREFIX}{message}")

with indent_print():
    custom_print("Level 1")
    with indent_print():
        custom_print("Level 2")
    custom_print("Back to Level 1")
# Output:
#   Level 1
#     Level 2
```

```
#   Back to Level 1
```

Exercise 6.2 Solution

```
from contextlib import contextmanager

@contextmanager
def read_data(filename):
    print(f"Opening file: {filename}")
    try:
        yield filename
    finally:
        print(f"Closing file: {filename}")

with read_data("config.ini") as f:
    print(f"Processing data from {f}")
# Output:
# Opening file: config.ini
# Processing data from config.ini
# Closing file: config.ini
```

Exercise 6.3 Solution

```
from contextlib import contextmanager

@contextmanager
def retry_on_fail(max_attempts):
    try:
        yield
    except Exception as e:
        print(f"Attempt failed with: {e}")
        # Mocking the retry logic here since the CM exits after the 'except'
        for attempt in range(2, max_attempts + 1):
            print(f"Remaining retries: {max_attempts - attempt + 1}. Printing pla
        # Re-raise the exception to exit the context normally after 'mock' retrie
        raise

try:
    with retry_on_fail(3):
        raise RuntimeError("Network issue.")
except RuntimeError as e:
    print(f"Outer block caught: {e.__class__.__name__}")
# Output:
# Attempt failed with: Network issue.
# Remaining retries: 2. Printing placeholder for logic.
# Remaining retries: 1. Printing placeholder for logic.
# Outer block caught: RuntimeError
```

Exercise 6.4 Solution

```
from contextlib import contextmanager
import time

@contextmanager
def time_block(label):
    start_time = time.time()
    try:
        yield
    finally:
        end_time = time.time()
        duration = end_time - start_time
        print(f"[{label}] finished in {duration:.4f} seconds.")

with time_block("DB_FETCH"):
    time.sleep(0.01)
# Output (approx):
# [DB_FETCH] finished in 0.01xx seconds.
```

Exercise 6.5 Solution

```
from contextlib import contextmanager

@contextmanager
def silent_fail():
    try:
        yield
    except Exception:
        print("Error suppressed silently.")
        # By not re-raising, the exception is swallowed/suppressed

with silent_fail():
    print("Pre-crash")
    1 / 0
    print("Post-crash (should not run)")

print("Execution continues outside the block.")
# Output:
# Pre-crash
# Error suppressed silently.
# Execution continues outside the block.
```

Exercise 6.6 Solution

```
from contextlib import contextmanager

@contextmanager
def suppress_unless(error_type_to_propagate):
    try:
        yield
    except error_type_to_propagate:
        print(f"PROPAGATING: {error_type_to_propagate.__name__}")
        raise # Reraise the specified error
    except Exception as e:
        print(f"SUPPRESSED: {e.__class__.__name__} was ignored.")
```

```
        # Suppress all others

# Test 1: Suppress (default behavior)
with suppress_unless(ValueError):
    raise IndexError("Ignored")
print("---")
# Test 2: Propagate (ValueError specified)
try:
    with suppress_unless(ValueError):
        raise ValueError("Propagate me")
except ValueError as e:
    print(f"Caught propagated error: {e.__class__.__name__}")
# Output:
# SUPPRESSED: IndexError was ignored.
# ---
# PROPAGATING: ValueError
# Caught propagated error: ValueError
```

Exercise 6.7 Solution

```
from contextlib import contextmanager

def is_non_empty_string(value):
    return isinstance(value, str) and len(value) > 0

@contextmanager
def validate_input(input_value, validation_func):
    if not validation_func(input_value):
        raise ValueError(f"Input '{input_value}' failed validation.")
    try:
        yield
    finally:
        pass # No cleanup needed

try:
    with validate_input("Valid string", is_non_empty_string):
        print("Block 1 runs.")
    with validate_input("", is_non_empty_string):
        print("Block 2 runs. (Should not reach)")
except ValueError as e:
    print(f"Caught validation error: {e}")
# Output:
# Block 1 runs.
# Caught validation error: Input '' failed validation.
```

Exercise 6.8 Solution

```
from contextlib import contextmanager
import threading

# Assume safe_lock is defined as in 5.8
@contextmanager
def safe_lock(lock):
    print("LOCK: Acquiring...")
    lock.acquire()
```

```
    try:
        yield
    finally:
        lock.release()
        print("LOCK: Released.")

@contextmanager
def resource_allocator(name):
    print(f"RESOURCE: Allocating {name}")
    try:
        yield f"ID_{name}"
    finally:
        print(f"RESOURCE: Deallocating {name}")

my_nested_lock = threading.Lock()

with safe_lock(my_nested_lock):
    with resource_allocator("A") as res_a:
        print(f"Working with {res_a} while locked.")
    print("Lock held during resource allocation/deallocation.")
# Output:
# LOCK: Acquiring...
# RESOURCE: Allocating A
# Working with ID_A while locked.
# RESOURCE: Deallocating A
# Lock held during resource allocation/deallocation.
# LOCK: Released.
```

Mixed Decorators + Context Managers Solutions

Exercise 7.1 Solution

```
from functools import wraps
from contextlib import contextmanager
import time

@contextmanager
def timer_cm():
    start_time = time.time()
    try:
        yield
    finally:
        end_time = time.time()
        duration = end_time - start_time
        print(f"Function executed in {duration:.4f} seconds.")

def with_timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # The key: using the CM inside the decorator's wrapper
        with timer_cm():
            return func(*args, **kwargs)
    return wrapper

@with_timer
def fetch_data():
    time.sleep(0.03)
    return "Data fetched."
```

```
print(fetch_data())
# Output (approx):
# Function executed in 0.03xx seconds.
# Data fetched.
```

Exercise 7.2 Solution

```
from functools import wraps
from contextlib import contextmanager
import threading
import time

@contextmanager
def safe_lock(lock):
    lock.acquire()
    try:
        yield
    finally:
        lock.release()

def with_lock(lock):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Use CM inside decorator
            with safe_lock(lock):
                return func(*args, **kwargs)
        return wrapper
    return decorator

SHARED_LOCK = threading.Lock()

@with_lock(SHARED_LOCK)
def update_shared_resource(value):
    print(f"Lock status inside function: {SHARED_LOCK.locked()}")
    time.sleep(0.01)
    return f"Updated with {value}"

print(update_shared_resource(100))
print(f"Lock status outside function: {SHARED_LOCK.locked()}")
# Output:
# Lock status inside function: True
# Updated with 100
# Lock status outside function: False
```

Exercise 7.3 Solution

```
from contextlib import contextmanager

GLOBAL_CACHE_MOCK = {}
GLOBAL_CALC_COUNT = 0

def original_compute_val(x):
    global GLOBAL_CALC_COUNT
```

```
    GLOBAL_CALC_COUNT += 1
    return x * 10

def cached_compute_val(x):
    if x not in GLOBAL_CACHE_MOCK:
        GLOBAL_CACHE_MOCK[x] = original_compute_val(x)
    return GLOBAL_CACHE_MOCK[x]

# Function holder (this is what the user calls)
compute_val = original_compute_val

@contextmanager
def temporary_cache():
    global compute_val
    original_func = compute_val
    compute_val = cached_compute_val # Switch to cached version
    try:
        yield
    finally:
        compute_val = original_func # Switch back

# Test 1: No cache
print(f"No Cache: {compute_val(1)} | Count: {GLOBAL_CALC_COUNT}")

# Test 2: Inside cache
with temporary_cache():
    print(f"In Cache: {compute_val(2)} | Count: {GLOBAL_CALC_COUNT}") # Calc 1
    print(f"In Cache: {compute_val(2)} | Count: {GLOBAL_CALC_COUNT}") # Cache hit

# Test 3: Back to no cache
print(f"No Cache: {compute_val(3)} | Count: {GLOBAL_CALC_COUNT}") # Calc 2
# Output:
# No Cache: 10 | Count: 1
# In Cache: 20 | Count: 2
# In Cache: 20 | Count: 2
# No Cache: 30 | Count: 3
```

Exercise 7.4 Solution

```
from functools import wraps
from contextlib import contextmanager

CURRENT_PREFIX = ""

@contextmanager
def state_prefix_manager(prefix):
    global CURRENT_PREFIX
    original_prefix = CURRENT_PREFIX
    CURRENT_PREFIX = prefix
    try:
        yield
    finally:
        CURRENT_PREFIX = original_prefix

def with_state_prefix(prefix):
    def decorator(func):
        @wraps(func)
```

```
        def wrapper(*args, **kwargs):
            # Use CM inside decorator
            with state_prefix_manager(prefix):
                return func(*args, **kwargs)
        return wrapper
    return decorator

def message(text):
    print(f"[{CURRENT_PREFIX}] {text}")

message("Initial message.")

@with_state_prefix("API")
def api_call(endpoint):
    message(f"Calling {endpoint}...")

@with_state_prefix("DB")
def db_query(sql):
    message(f"Executing {sql}...")

api_call("/users")
db_query("SELECT *")
message("Final message.")
# Output:
# [] Initial message.
# [API] Calling /users...
# [DB] Executing SELECT *...
# [] Final message.
```

Exercise 7.5 Solution

```
from contextlib import contextmanager
from functools import wraps

# Simple logging decorator for demonstration
def simple_log(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("LOGGED EXECUTION.")
        return func(*args, **kwargs)
    return wrapper

@contextmanager
def decorate_block(decorator):
    print("CM: Applying decorator...")
    def mock_function():
        return "Original Call."

    # Decorate the mock function
    decorated_func = decorator(mock_function)

    try:
        yield decorated_func
    finally:
        print("CM: Context exited.")

# Usage: Capture the decorated function as 'action'
with decorate_block(simple_log) as action:
```

```
    result = action()
    print(result)
# Output:
# CM: Applying decorator...
# LOGGED EXECUTION.
# Original Call.
# CM: Context exited.
```

Paragraphic / Scenario-based Hard Exercises Solutions

Exercise 8.1 Solution: API Call Timer and Logger

```
from functools import wraps
import time

def monitor_api_call(service_name):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # 1. Log call
            print(f"CALL [{service_name}]: {func.__name__} called with args={args

            # 2. Timing
            start_time = time.time()
            result = func(*args, **kwargs)
            end_time = time.time()
            duration = end_time - start_time

            # 3. Log time
            print(f"TIME [{service_name}]: {func.__name__} took {duration:.4f} se

            # 4. Log return
            print(f"RETURN [{service_name}]: {result}")

            return result
        return wrapper
    return decorator

@monitor_api_call("UserAuth")
def fetch_user(user_id, verbose=False):
    time.sleep(0.01)
    if user_id == 1:
        return {"id": 1, "name": "Alice"}
    return {"id": user_id, "name": "Guest"}

user_data = fetch_user(1, verbose=True)
print(f"Function Result: {user_data}")
```

Exercise 8.2 Solution: Database Transaction Context Manager (Mock)

```
from contextlib import contextmanager

@contextmanager
def db_transaction(db_conn):
```

```
    print(f"Transaction BEGIN for {db_conn}")
    try:
        yield db_conn
        print("Transaction COMMIT")
    except Exception as e:
        print("Transaction ROLLBACK")
        # Re-raise the original exception
        raise e

try:
    with db_transaction("Postgres_PROD") as db:
        print(f"Working with {db}...")
        result = 10 / 0 # Force an exception
        print(result) # Should not run
except ZeroDivisionError as e:
    print(f"Caught external error: {e.__class__.__name__}")
```

Exercise 8.3 Solution: Access Control with Default Role Factory

```
from functools import wraps

# Global state
USER_PERMISSIONS = {'delete_data': 'guest', 'read_data': 'admin'}

def restrict_to(required_role):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            func_name = func.__name__
            current_role = USER_PERMISSIONS.get(func_name, 'unknown')

            if current_role != required_role:
                raise PermissionError(
                    f"Access Denied: Function '{func_name}' requires role '{requi
                )
            return func(*args, **kwargs)
        return wrapper
    return decorator

@restrict_to('admin')
def delete_data():
    return "Data wiped."

@restrict_to('guest')
def read_data():
    return "Data read."

print(read_data())
try:
    delete_data()
except PermissionError as e:
    print(e)
```

Exercise 8.4 Solution: Read-Only State Context Manager

```
from contextlib import contextmanager

CAN_WRITE = True

@contextmanager
def read_only_mode():
    global CAN_WRITE
    original_state = CAN_WRITE

    # Setup
    CAN_WRITE = False
    print(f"Entered R/O mode. CAN_WRITE={CAN_WRITE}")

    try:
        yield
    except Exception as e:
        # Ensures teardown runs even on exception
        raise e
    finally:
        # Restore ONLY if the original state was True, or just always restore it
        CAN_WRITE = original_state # Restore to initial state
        print(f"Exited R/O mode. CAN_WRITE={CAN_WRITE}")

print(f"Initial: {CAN_WRITE}")

with read_only_mode():
    with read_only_mode(): # Nested entry
        print(f"Nested Check: {CAN_WRITE}")
    print(f"Back to Outer: {CAN_WRITE}")

print(f"Final: {CAN_WRITE}")
```

Exercise 8.5 Solution: Combined Validation and Logging Decorator

```
from functools import wraps

def log_and_validate(func):
    @wraps(func)
    def wrapper(value, *args, **kwargs):
        # 1. Validation
        if not (isinstance(value, (int, float)) and value >= 0):
            raise ValueError(
                f"Validation Failed: First argument must be a non-negative number
            )

        # Execution
        result = func(value, *args, **kwargs)

        # 2. Logging
        print(f"LOG: {func.__name__} executed successfully. Result: {result}")

        return result
    return wrapper

@log_and_validate
def process_amount(amount, tax_rate=0.1):
```

```
    return amount * (1 + tax_rate)

print(process_amount(100))
try:
    process_amount(-50)
except ValueError as e:
    print(f"Caught Error: {e}")
```

Exercise 8.6 Solution: Custom Error Mapping Context Manager (Factory)

```
from contextlib import contextmanager

def map_error(original_error, target_error):
    @contextmanager
    def error_mapper():
        try:
            yield
        except original_error as e:
            # Catch the original error and raise the target error
            raise target_error(f"Error mapped from {original_error.__name__}: {e}
    return error_mapper

# Usage
mapper = map_error(ZeroDivisionError, ValueError)

try:
    with mapper():
        print("Mapping block...")
        1 / 0
except ValueError as e:
    print(f"Successfully caught mapped error: {e.__class__.__name__} - {e}")
except Exception as e:
    print(f"Caught other error: {e.__class__.__name__}")
```

Exercise 8.7 Solution: Decorator for Argument Type Checking (Multiple)

```
from functools import wraps

def check_types(int_args=[], str_args=[]):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check integer arguments
            for arg_name in int_args:
                if arg_name in kwargs and not isinstance(kwargs[arg_name], int):
                    raise TypeError(
                        f"Argument '{arg_name}' must be an integer (got {type(kwa
                    )
            # Check string arguments
            for arg_name in str_args:
                if arg_name in kwargs and not isinstance(kwargs[arg_name], str):
                    raise TypeError(
                        f"Argument '{arg_name}' must be a string (got {type(kwarg
                    )
```

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

@check_types(int_args=['age', 'id'], str_args=['name', 'department'])
def create_profile(name, age, id, department="IT"):
    return f"Profile: {name}, Age: {age}, ID: {id}, Dept: {department}"

print(create_profile(name="Sam", age=30, id=101))
try:
    create_profile(name="Sam", age="30", id=102) # Error: age is str
except TypeError as e:
    print(f"Caught Error: {e}")
```

Exercise 8.8 Solution: Dynamic Resource Yielding Context Manager

```
from contextlib import contextmanager

@contextmanager
def dynamic_resource_cm(config_name):
    # Resource is a mutable dictionary
    resource = { 'status': 'READY', 'config': config_name }
    print(f"Setup: Resource {config_name} initialized.")

    try:
        yield resource
    except Exception as e:
        resource['status'] = 'CLOSED_WITH_ERROR'
        raise e
    finally:
        # Cleanup: Modify the yielded object in the 'finally' block
        if resource['status'] == 'READY':
            resource['status'] = 'CLOSED'
        print(f"Teardown: Resource {config_name} status is now {resource['status'

# Usage
res_a = None
with dynamic_resource_cm("System_A") as res:
    print(f"Inside: Status={res['status']}")
    res_a = res

print(f"Outside: Final Status={res_a['status']}")
```

Exercise 8.9 Solution: Sequential Decorator and Context Manager Interaction

```
from functools import wraps
from contextlib import contextmanager
import time

# Decorator from 2.8
def time_it(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"TIME_IT: START {func.__name__}")
```

```
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        duration = end_time - start_time
        print(f"TIME_IT: {func.__name__} took {duration:.4f} seconds.")
        return result
    return wrapper

# Context Manager
@contextmanager
def suppress_io_errors():
    print("CM: Setup - Suppression active.")
    try:
        yield
    except IOError:
        print("CM: IOError suppressed.")
    except Exception as e:
        print(f"CM: Propagating {e.__class__.__name__}.")
        raise e
    finally:
        print("CM: Teardown - Suppression deactivated.")

@time_it # Outer/First-applied decorator
def calculate(a, b):
    print(f"Function: Calculating {a} + {b}...")
    time.sleep(0.01)
    raise TimeoutError("Calculation took too long.")
    return a + b

try:
    with suppress_io_errors():
        result = calculate(10, 20)
        print(result) # Should not run
except TimeoutError as e:
    print(f"EXTERNAL: Caught propagated error: {e.__class__.__name__}")

# Execution Trace:
# 1. Outer CM Setup: CM: Setup - Suppression active.
# 2. Decorator Wrapper: TIME_IT: START calculate
# 3. Function Execution: Function: Calculating 10 + 20...
# 4. Exception (TimeoutError) Raised.
# 5. CM Exception Handler: Catches TimeoutError, sees it's not IOError, and re-ra
# 6. CM Teardown: CM: Teardown - Suppression deactivated.
# 7. Decorator Teardown (Finally): TIME_IT logs time.
# 8. External Handler: Catches TimeoutError.
```

Exercise 8.10 Solution: Conditional Caching Decorator Factory

```
from functools import wraps

CONDITIONAL_CACHE = {}

def cache_if(condition_arg_name):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            use_cache = kwargs.get(condition_arg_name, False)
```

```
            # Create a cache key independent of the condition_arg_name
            clean_kwargs = {k: v for k, v in kwargs.items() if k != condition_arg
            cache_key = (func.__name__, args, tuple(sorted(clean_kwargs.items())))

            if use_cache and cache_key in CONDITIONAL_CACHE:
                print(f"CACHE HIT for {func.__name__}")
                return CONDITIONAL_CACHE[cache_key]

            print(f"Calculating {func.__name__}...")
            result = func(*args, **kwargs)

            if use_cache:
                CONDITIONAL_CACHE[cache_key] = result
                print("CACHE STORED.")

            return result
        return wrapper
    return decorator

@cache_if('use_cache')
def generate_report(month, year, use_cache=False):
    return f"Report for {month}/{year} (Generated at {time.time():.2f})"

print(generate_report("Jan", 2024, use_cache=True))
print(generate_report("Jan", 2024, use_cache=True)) # Cache Hit
print(generate_report("Feb", 2024, use_cache=False)) # Calculates, does not cache
print(generate_report("Feb", 2024, use_cache=True)) # Calculates and caches
```

Exercise 8.11 Solution: Nested Lock with Timeout Context Manager (Mock)

```
from contextlib import contextmanager
import threading
import time

@contextmanager
def resource_lock_with_timeout(lock, timeout):
    # Mocking a check on a lock that might be acquired by another thread
    if lock.locked():
        raise TimeoutError(f"Failed to acquire lock within {timeout}s.")

    print(f"Attempting to acquire lock with timeout={timeout}...")

    # In a real scenario, this would be lock.acquire(timeout=timeout)
    lock.acquire()

    try:
        # Simulate work
        yield
        time.sleep(0.01)
    finally:
        lock.release()
        print("Lock released.")

my_lock = threading.Lock()

# Test 1: Success
```

```
with my_lock: # Acquire lock outside
    try:
        with resource_lock_with_timeout(my_lock, 0.5): # This should raise Timeou
            pass
    except TimeoutError as e:
        print(f"Caught expected error: {e}")
# Output:
# Caught expected error: Failed to acquire lock within 0.5s.
# Lock released. (This is the CM's release, the outer 'with' releases after)
# Lock released. (This is the outer 'with' block's implicit release)
```

Exercise 8.12 Solution: Debugger Mode Context Manager

```
from contextlib import contextmanager

IS_DEBUG = False

@contextmanager
def debug_mode():
    global IS_DEBUG
    original_state = IS_DEBUG

    IS_DEBUG = True
    print("DEBUG CM: Enabled.")

    try:
        yield IS_DEBUG # Yield the boolean value (True)
    except Exception as e:
        raise e
    finally:
        IS_DEBUG = original_state
        print(f"DEBUG CM: Disabled. IS_DEBUG={IS_DEBUG}")

print(f"Initial Debug: {IS_DEBUG}")

with debug_mode() as debug_flag:
    print(f"Inside Block: Debug Flag is {debug_flag}")
    if debug_flag:
        print("This is a debug message.")
    else:
        print("This should not print.")

print(f"Final Debug: {IS_DEBUG}")
```

Exercise 8.13 Solution: Multi-Stage Validation Decorator

```
from functools import wraps

def check_is_even(n):
    return n % 2 == 0

def check_is_in_range(n):
    return 10 <= n <= 20
```

```
def multi_stage_validation(arg_name, check_funcs):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            value = kwargs.get(arg_name)

            for check_func in check_funcs:
                if not check_func(value):
                    raise ValueError(
                        f"Validation Failed for argument '{arg_name}'. Check '{ch
                    )
            return func(*args, **kwargs)
        return wrapper
    return decorator

@multi_stage_validation('num', [check_is_even, check_is_in_range])
def process_number(num):
    return f"Number {num} passed all checks."

print(process_number(num=18)) # Success
try:
    process_number(num=15) # Fails check_is_even
except ValueError as e:
    print(f"Caught Error: {e}")
```

Exercise 8.14 Solution: Environment Variable Context Manager

```
from contextlib import contextmanager

ENV_VARS = {'PATH': '/usr/bin', 'DEBUG': 'False'}

@contextmanager
def temp_environment(key, value):
    global ENV_VARS
    original_value = ENV_VARS.get(key) # Get original or None
    key_existed = key in ENV_VARS

    print(f"Setting {key} = {value}")
    ENV_VARS[key] = value

    try:
        yield
    except Exception as e:
        raise e
    finally:
        # Restore logic
        if key_existed:
            ENV_VARS[key] = original_value
        else:
            del ENV_VARS[key]
        print(f"Restored ENV. {key} is now {ENV_VARS.get(key, 'DELETED')}")

# Test 1: Existing key
with temp_environment('DEBUG', 'True'):
    print(f"Inside: DEBUG={ENV_VARS['DEBUG']}")

# Test 2: New key
```

```
with temp_environment('NEW_VAR', 'temp'):
    print(f"Inside: NEW_VAR={ENV_VARS['NEW_VAR']}")
```

Exercise 8.15 Solution: Decorator that Enforces Context Manager Usage

```
from functools import wraps

# Mock Resource Class (simulates what a CM would create/manage)
class MockConnection:
    def __init__(self, is_active=False):
        self.is_active = is_active # State set by the CM

def requires_context_manager(func):
    @wraps(func)
    def wrapper(connection_object, *args, **kwargs):
        # Check if the connection object has the required active state
        if not hasattr(connection_object, 'is_active') or not connection_object.i
            raise RuntimeError(
                f"Connection for '{func.__name__}' is not active. "
                f"Did you forget to use a Context Manager (the 'with' block)?"
            )
        return func(connection_object, *args, **kwargs)
    return wrapper

@requires_context_manager
def process_data(conn):
    return f"Processing successful with active connection: {conn.is_active}"

# Test 1: Fails (not active)
conn_inactive = MockConnection(is_active=False)
try:
    process_data(conn_inactive)
except RuntimeError as e:
    print(f"Caught Error: {e}")

# Test 2: Success (simulated active by a CM)
conn_active = MockConnection(is_active=True)
print(process_data(conn_active))
```

Conceptual Callouts (Tricky Parts)

1. Closure Capture & Late Binding

- **What it is:** When a function (the inner `wrapper` in a decorator) references a variable from its enclosing scope (like `prefix` in a decorator factory), it creates a **closure**.
- **Common Mistake:** In loops creating multiple functions, the closure variables are often not captured uniquely for each iteration. They all reference the *same* variable, leading to "late binding" where all functions use the variable's final value, not the value it had when the function was defined.
- **Solution (in this workbook):** This issue is naturally avoided in the factory pattern (`def factory(arg): def decorator(func):...`) because the factory argument (`arg`) is

captured uniquely by the `decorator` function for each decorated call.

2. Why `functools.wraps` Matters

- **Problem:** Without `@wraps(func)`, the decorated function (e.g., `calculate_sum`) takes on the metadata of its wrapper function (e.g., `wrapper`). This means `calculate_sum.__name__` becomes `'wrapper'`, and `calculate_sum.__doc__` becomes `None`.
- **Why it's Crucial:** This loss of metadata breaks debugging tools, documentation generators (like Sphinx), and any code that uses reflection (e.g., function logging, caching keys based on name).
- **Solution:** `@functools.wraps(func)` copies the original function's name, docstring, module, and dictionary, preserving its identity.

3. Decorator Execution Order

- **Rule:** Decorators are applied and executed from **bottom to top**, or **inner to outer**.

```
@decorator_outer  # Called second, wraps the result of decorator_inner
@decorator_inner  # Called first, wraps the original function
def my_func(): ...
```

- **Result:** When `my_func()` is called, the execution flow runs through `decorator_outer`'s wrapper first, then through `decorator_inner`'s wrapper, and finally executes the original `my_func`. The teardown/post-execution logic then runs in the reverse order (inner wrapper, then outer wrapper).

4. How `with` Translates to `__enter__` / `__exit__`

- **Concept (Class-based):** The `with` statement requires the context manager object to have two methods:
  1. `__enter__(self)`: Called upon entering the block. Its return value is bound to the `as` variable (if present).
  2. `__exit__(self, exc_type, exc_val, exc_tb)`: Called upon exiting the block (teardown), *even if an exception occurred*. If an exception occurred, the exception