Pemrograman Berbasis Fungsi - Minggu 11

Teknik Decorator Design Pattern

Tujuan Pembelajaran

Mahasiswa mampu memahami dan mengimplementasikan pola desain decorator dalam konteks pemrograman fungsional untuk menambahkan fungsionalitas pada fungsi secara dinamis tanpa mengubah struktur aslinya.

Topik Bahasan

1. Konsep Decorator Pattern dalam Functional Programming
2. Implementasi Decorator menggunakan Python
3. Function Wrapping dan Closure
4. Decorator dengan Parameter
5. Studi Kasus: Logging, Timing, dan Authentication

Dosen: Ahmad Luky Ramdani
Program Studi: Sains Data
Institut Teknologi Sumatera

1. Konsep Decorator Pattern dalam Functional Programming

Apa itu Decorator?

Decorator adalah pola desain (design pattern) yang memungkinkan kita untuk **menambahkan fungsionalitas baru** pada sebuah fungsi atau objek **tanpa mengubah struktur aslinya**.

Analogi Sederhana

Bayangkan Anda membeli sebuah **kado**: - Kado asli = Fungsi asli - Bungkus kado = Decorator pertama - Pita dan kartu ucapan = Decorator kedua

Anda tetap memberikan kado yang sama, tetapi dengan **tampilan yang lebih menarik** tanpa mengubah isi kadonya!

Mengapa Menggunakan Decorator?

✅ **Reusable**: Dapat digunakan berulang kali pada fungsi yang berbeda

✅ **Clean Code**: Kode lebih rapi dan terorganisir

✅ **Separation of Concerns**: Memisahkan logika utama dari logika tambahan

✅ **Dynamic**: Menambahkan fungsionalitas secara dinamis saat runtime

Kapan Menggunakan Decorator?

- Logging (mencatat aktivitas)
- Timing (mengukur waktu eksekusi)
- Authentication/Authorization (validasi akses)
- Caching (menyimpan hasil komputasi)
- Input validation (validasi parameter)
- Error handling

2. Implementasi Decorator Menggunakan Python

2.1 Contoh Tanpa Decorator (Cara Manual)

Mari kita lihat contoh fungsi sederhana tanpa menggunakan decorator:

```
def sapa(nama):
    """Fungsi untuk menyapa seseorang"""
    return f"Halo, {nama}!"

# Memanggil fungsi
hasil = sapa("Budi")
print(hasil)
```

```
Halo, Budi!
```

Sekarang, bagaimana jika kita ingin menambahkan **garis pembatas** di atas dan di bawah sapaan? Tanpa decorator, kita harus mengubah fungsi aslinya:

```
def sapa_dengan_garis(nama):
    """Fungsi untuk menyapa dengan garis pembatas"""
    print("="*30)
    hasil = f"Halo, {nama}!"
    print(hasil)
    print("="*30)
    return hasil

sapa_dengan_garis("Ani")
```

```
==============================
Halo, Ani!
==============================
```

```
'Halo, Ani!'
```

Masalah dengan pendekatan ini: - Kita mengubah fungsi asli - Tidak reusable (harus copy-paste untuk fungsi lain) - Code duplication

Solusi: Gunakan Decorator!

2.2 Decorator Sederhana

Decorator adalah fungsi yang **menerima fungsi lain sebagai parameter** dan **mengembalikan fungsi baru** yang telah dimodifikasi.

```
def decorator_garis(func):
    """
    Decorator yang menambahkan garis pembatas

    Args:
        func: Fungsi yang akan di-decorate

    Returns:
        wrapper: Fungsi baru yang telah dimodifikasi
    """
    def wrapper(*args, **kwargs):
        # Kode yang dijalankan SEBELUM fungsi asli
        print("="*30)

        # Memanggil fungsi asli
        hasil = func(*args, **kwargs)

        # Kode yang dijalankan SETELAH fungsi asli
        print("="*30)

        return hasil

    return wrapper

# Fungsi asli (tanpa modifikasi)
def sapa(nama):
    return f"Halo, {nama}!"

# Menerapkan decorator secara manual
sapa_dengan_dekorasi = decorator_garis(sapa)
print(sapa_dengan_dekorasi("Citra"))
```

```
==============================
==============================
Halo, Citra!
```

2.3 Menggunakan Syntax @ (Syntactic Sugar)

Python menyediakan syntax `@` yang lebih elegan untuk mengaplikasikan decorator:

```python
# Menggunakan @ syntax
@decorator_garis
def sapa_formal(nama, gelar="Saudara/i"):
    return f"Halo, {gelar} {nama}!"

# Langsung bisa dipanggil
print(sapa_formal("Dewi", "Dr."))
```

```
==============================
==============================
Halo, Dr. Dewi!
```

Penjelasan:

```python
@decorator_garis
def sapa_formal(nama, gelar="Saudara/i"):
    ...
```

Ekuivalen dengan:

```python
def sapa_formal(nama, gelar="Saudara/i"):
    ...
sapa_formal = decorator_garis(sapa_formal)
```

Syntax `@` membuat kode lebih **bersih** dan **mudah dibaca**!

Latihan 1: Buat Decorator Sederhana

Tugas: Buat decorator bernama `decorator_bintang` yang menambahkan bintang (★) di sebelah kiri dan kanan output fungsi.

Contoh output yang diharapkan:

```
★★★★★ Halo, Budi! ★★★★★
```

```python
# TULIS KODE ANDA DI SINI
def decorator_bintang(func):
    # TODO: Lengkapi decorator ini
    pass

@decorator_bintang
def sapa(nama):
    return f"Halo, {nama}!"

# Test
# print(sapa("Budi"))
```

3. Function Wrapping dan Closure

3.1 Apa itu Closure?

Closure adalah fungsi yang dapat mengakses variabel dari scope luar (outer function) meskipun outer function sudah selesai dieksekusi.

Visualisasi Closure:

```
def fungsi_luar(pesan):
    """Outer function"""
    # Variabel di scope luar
    prefix = "INFO: "

    def fungsi_dalam(nama):
        """Inner function - ini adalah closure"""
        # Mengakses variabel dari fungsi luar
        return f"{prefix}{pesan} {nama}!"

    return fungsi_dalam

# Membuat closure
sapa_info = fungsi_luar("Halo")
print(sapa_info("Eko"))

# Membuat closure lain dengan pesan berbeda
sapa_warning = fungsi_luar("Peringatan untuk")
print(sapa_warning("Eko"))
```

```
INFO: Halo Eko!
INFO: Peringatan untuk Eko!
```

Penjelasan Alur:

1. `fungsi_luar("Halo")` dipanggil → `pesan = "Halo"`
2. `fungsi_dalam` dibuat dan **mengingat** variabel `pesan` dan `prefix`
3. `fungsi_luar` selesai, tetapi `fungsi_dalam` masih bisa mengakses `pesan` dan `prefix`
4. Saat `sapa_info("Eko")` dipanggil, closure menggunakan nilai yang diingat

3.2 Decorator Menggunakan Closure

Decorator memanfaatkan konsep closure untuk "mengingat" fungsi asli:

```
def decorator_uppercase(func):
    """Decorator yang mengubah hasil fungsi menjadi uppercase"""

    def wrapper(*args, **kwargs):
        # Memanggil fungsi asli (yang "diingat" oleh closure)
        hasil = func(*args, **kwargs)
```

```
        # Modifikasi hasil
        return hasil.upper()

    return wrapper

@decorator_uppercase
def sapa(nama):
    return f"Halo, {nama}!"

print(sapa("fandi"))  # Output: HALO, FANDI!
```

```
HALO, FANDI!
```

3.3 Menggunakan functools.wraps

Masalah: Decorator mengubah metadata fungsi asli (nama, docstring, dll).

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def sapa(nama):
    """Fungsi untuk menyapa seseorang"""
    return f"Halo, {nama}!"

# Lihat metadata fungsi
print(f"Nama fungsi: {sapa.__name__}")  # wrapper, bukan sapa!
print(f"Docstring: {sapa.__doc__}")     # None, bukan docstring asli!
```

```
Nama fungsi: wrapper
Docstring: None
```

Solusi: Gunakan `functools.wraps` untuk mempertahankan metadata:

```
from functools import wraps

def my_decorator_fixed(func):
    @wraps(func)  # Mempertahankan metadata fungsi asli
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@my_decorator_fixed
```

```python
def sapa_fixed(nama):
    """Fungsi untuk menyapa seseorang"""
    return f"Halo, {nama}!"

# Lihat metadata fungsi
print(f"Nama fungsi: {sapa_fixed.__name__}")  # sapa_fixed ✓
print(f"Docstring: {sapa_fixed.__doc__}")     # Fungsi untuk menyapa seseorang
✓
```

```
Nama fungsi: sapa_fixed
Docstring: Fungsi untuk menyapa seseorang
```

💡 **Best Practice:** Selalu gunakan `@wraps(func)` dalam decorator Anda!

4. Decorator dengan Parameter

4.1 Mengapa Perlu Decorator dengan Parameter?

Terkadang kita ingin **mengkustomisasi** perilaku decorator. Contoh: - Decorator logging dengan level berbeda (INFO, WARNING, ERROR) - Decorator repeat dengan jumlah pengulangan berbeda - Decorator validation dengan rules berbeda

4.2 Struktur Decorator dengan Parameter

Decorator dengan parameter membutuhkan **3 level fungsi**:

```
Fungsi Level 1: Menerima parameter decorator
    └─ Fungsi Level 2: Menerima fungsi yang di-decorate
        └─ Fungsi Level 3: Wrapper yang memanggil fungsi asli
```

```python
from functools import wraps

def repeat(jumlah):
    """
    Decorator dengan parameter untuk mengulang eksekusi fungsi

    Args:
        jumlah: Berapa kali fungsi akan diulang
    """
    # Level 1: Menerima parameter decorator
    def decorator(func):
        # Level 2: Menerima fungsi yang di-decorate
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Level 3: Wrapper yang memanggil fungsi asli
            hasil = None
            for i in range(jumlah):
```

```
                print(f"Eksekusi ke-{i+1}:")
                hasil = func(*args, **kwargs)
            return hasil
        return wrapper
    return decorator

# Menggunakan decorator dengan parameter
@repeat(jumlah=3)
def sapa(nama):
    print(f"Halo, {nama}!")
    return "Selesai"

hasil = sapa("Gita")
```

```
Eksekusi ke-1:
Halo, Gita!
Eksekusi ke-2:
Halo, Gita!
Eksekusi ke-3:
Halo, Gita!
```

4.3 Decorator dengan Multiple Parameters

Kita bisa menambahkan lebih dari satu parameter:

```
from functools import wraps

def border(karakter="=", panjang=30):
    """
    Decorator dengan multiple parameter untuk menambahkan border

    Args:
        karakter: Karakter untuk border (default: =)
        panjang: Panjang border (default: 30)
    """
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(karakter * panjang)
            hasil = func(*args, **kwargs)
            print(karakter * panjang)
            return hasil
        return wrapper
    return decorator

# Menggunakan parameter default
@border()
```

```python
def fungsi1():
    print("Fungsi dengan border default")

fungsi1()
print()

# Menggunakan custom parameter
@border(karakter="*", panjang=50)
def fungsi2():
    print("Fungsi dengan border custom")

fungsi2()
```

```
==============================
Fungsi dengan border default
==============================

**************************************************
Fungsi dengan border custom
**************************************************
```

Latihan 2: Decorator dengan Parameter

Tugas: Buat decorator bernama `prefix_output` yang menambahkan prefix pada output fungsi.

Contoh penggunaan:

```python
@prefix_output(prefix="[INFO]")
def sapa(nama):
    return f"Halo, {nama}!"

print(sapa("Hani"))  # Output: [INFO] Halo, Hani!
```

```python
# TULIS KODE ANDA DI SINI
def prefix_output(prefix):
    # TODO: Lengkapi decorator ini
    pass

# @prefix_output(prefix="[INFO]")
# def sapa(nama):
#     return f"Halo, {nama}!"

# Test
# print(sapa("Hani"))
```

5. Studi Kasus: Aplikasi Decorator dalam Dunia Nyata

5.1 Studi Kasus 1: Logging Decorator

Logging adalah salah satu use case paling umum untuk decorator. Mari kita buat logging system yang profesional:

```
from functools import wraps
from datetime import datetime

def log_function_call(level="INFO"):
    """
    Decorator untuk logging function call dengan timestamp

    Args:
        level: Level logging (INFO, WARNING, ERROR)
    """
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Timestamp
            timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

            # Log sebelum eksekusi
            print(f"[{level}] [{timestamp}] Memanggil fungsi:
{func.__name__}")
            print(f"[{level}] Arguments: args={args}, kwargs={kwargs}")

            # Eksekusi fungsi
            try:
                hasil = func(*args, **kwargs)
                print(f"[{level}] [{timestamp}] Fungsi {func.__name__}
selesai")
                print(f"[{level}] Return value: {hasil}")
                return hasil
            except Exception as e:
                print(f"[ERROR] [{timestamp}] Error di fungsi {func.__name__}:
{str(e)}")
                raise

        return wrapper
    return decorator

# Contoh penggunaan
@log_function_call(level="INFO")
def hitung_luas_persegi(sisi):
    """Menghitung luas persegi"""
    return sisi * sisi
```

```python
@log_function_call(level="WARNING")
def bagi(a, b):
    """Membagi dua angka"""
    return a / b

# Test
print("=" * 60)
luas = hitung_luas_persegi(5)
print("=" * 60)
print()

hasil_bagi = bagi(10, 2)
print("=" * 60)
```

```
============================================================
[INFO] [2026-05-03 21:56:39] Memanggil fungsi: hitung_luas_persegi
[INFO] Arguments: args=(5,), kwargs={}
[INFO] [2026-05-03 21:56:39] Fungsi hitung_luas_persegi selesai
[INFO] Return value: 25
============================================================

[WARNING] [2026-05-03 21:56:39] Memanggil fungsi: bagi
[WARNING] Arguments: args=(10, 2), kwargs={}
[WARNING] [2026-05-03 21:56:39] Fungsi bagi selesai
[WARNING] Return value: 5.0
============================================================
```

5.2 Studi Kasus 2: Timing/Performance Decorator

Mengukur waktu eksekusi fungsi sangat penting untuk optimasi performa:

```python
import time
from functools import wraps

def measure_time(func):
    """Decorator untuk mengukur waktu eksekusi fungsi"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Catat waktu mulai
        start_time = time.time()

        # Eksekusi fungsi
        hasil = func(*args, **kwargs)

        # Catat waktu selesai
        end_time = time.time()
```

```
        # Hitung durasi
        durasi = end_time - start_time

        print(f"\n  Fungsi '{func.__name__}' selesai dalam {durasi:.4f}
detik")

        return hasil
    return wrapper

# Contoh: Fungsi yang lambat
@measure_time
def hitung_fibonacci(n):
    """Menghitung bilangan Fibonacci ke-n (recursive, tidak efisien)"""
    if n <= 1:
        return n
    return hitung_fibonacci.__wrapped__(n-1) +
hitung_fibonacci.__wrapped__(n-2)

# Contoh: Fungsi dengan delay
@measure_time
def proses_data(data):
    """Simulasi proses data yang memakan waktu"""
    print(f"Memproses {len(data)} item...")
    time.sleep(1)  # Simulasi delay
    return [x * 2 for x in data]

# Test
print("Test 1: Fibonacci")
fib = hitung_fibonacci(10)
print(f"Hasil: {fib}")

print("\nTest 2: Proses Data")
data = list(range(100))
hasil = proses_data(data)
print(f"Hasil: {hasil[:5]}...")  # Tampilkan 5 elemen pertama
```

```
Test 1: Fibonacci

  Fungsi 'hitung_fibonacci' selesai dalam 0.0000 detik
Hasil: 55

Test 2: Proses Data
Memproses 100 item...

  Fungsi 'proses_data' selesai dalam 1.0008 detik
Hasil: [0, 2, 4, 6, 8]...
```

5.3 Studi Kasus 3: Authentication/Authorization Decorator

Decorator untuk memvalidasi akses user sebelum menjalankan fungsi:

```
from functools import wraps

# Simulasi database user
USERS = {
    "admin": {"password": "admin123", "role": "admin"},
    "user1": {"password": "user123", "role": "user"},
    "user2": {"password": "user456", "role": "user"}
}

# Simulasi user yang sedang login
current_user = None

def login(username, password):
    """Fungsi untuk login"""
    global current_user
    if username in USERS and USERS[username]["password"] == password:
        current_user = {"username": username, "role": USERS[username]["role"]}
        print(f"✓ Login berhasil sebagai {username} (role:
{current_user['role']})")
        return True
    print("✗ Login gagal: Username atau password salah")
    return False

def logout():
    """Fungsi untuk logout"""
    global current_user
    if current_user:
        print(f"✓ Logout berhasil untuk {current_user['username']}")
        current_user = None
    else:
        print("✗ Tidak ada user yang sedang login")

def require_login(func):
    """Decorator yang memastikan user sudah login"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        if current_user is None:
            print(f"Akses ditolak! Anda harus login terlebih dahulu untuk
mengakses '{func.__name__}'")
            return None
        return func(*args, **kwargs)
    return wrapper

def require_role(role):
    """Decorator yang memastikan user memiliki role tertentu"""
```

```
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if current_user is None:
                print(f"Akses ditolak! Anda harus login terlebih dahulu")
                return None

            if current_user["role"] != role:
                print(f"Akses ditolak! Fungsi '{func.__name__}' hanya untuk
role '{role}'")
                print(f"   Role Anda: '{current_user['role']}'")
                return None

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

# Fungsi yang memerlukan login
@require_login
def lihat_profil():
    """Melihat profil user"""
    print(f" Profil Anda:")
    print(f"   Username: {current_user['username']}")
    print(f"   Role: {current_user['role']}")

# Fungsi yang hanya bisa diakses admin
@require_role("admin")
def hapus_user(username):
    """Menghapus user (hanya admin)"""
    if username in USERS:
        del USERS[username]
        print(f"User '{username}' berhasil dihapus")
    else:
        print(f"User '{username}' tidak ditemukan")

# Fungsi public (tidak perlu login)
def lihat_info_public():
    """Informasi yang bisa dilihat semua orang"""
    print("Informasi Public: Aplikasi versi 1.0")

# === SKENARIO TESTING ===
print("=" * 60)
print("SKENARIO 1: Akses tanpa login")
print("=" * 60)
lihat_profil()
hapus_user("user2")
lihat_info_public()
```

```python
print("\n" + "=" * 60)
print("SKENARIO 2: Login sebagai user biasa")
print("=" * 60)
login("user1", "user123")
lihat_profil()
hapus_user("user2")  # Tidak boleh akses

print("\n" + "=" * 60)
print("SKENARIO 3: Login sebagai admin")
print("=" * 60)
logout()
login("admin", "admin123")
lihat_profil()
hapus_user("user2")  # Boleh akses

print("\n" + "=" * 60)
```

```
============================================================
SKENARIO 1: Akses tanpa login
============================================================
Akses ditolak! Anda harus login terlebih dahulu untuk mengakses 'lihat_profil'
Akses ditolak! Anda harus login terlebih dahulu
Informasi Public: Aplikasi versi 1.0

============================================================
SKENARIO 2: Login sebagai user biasa
============================================================
✓ Login berhasil sebagai user1 (role: user)
 Profil Anda:
   Username: user1
   Role: user
Akses ditolak! Fungsi 'hapus_user' hanya untuk role 'admin'
   Role Anda: 'user'

============================================================
SKENARIO 3: Login sebagai admin
============================================================
✓ Logout berhasil untuk user1
✓ Login berhasil sebagai admin (role: admin)
 Profil Anda:
   Username: admin
   Role: admin
User 'user2' berhasil dihapus

============================================================
```

5.4 Studi Kasus 4: Caching/Memoization Decorator

Menyimpan hasil komputasi untuk meningkatkan performa:

```
import time
from functools import wraps

def cache(func):
    """
    Decorator untuk caching hasil fungsi
    Menyimpan hasil komputasi berdasarkan parameter input
    """
    # Dictionary untuk menyimpan cache
    cached_results = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        # Buat key dari arguments
        # Convert kwargs ke tuple agar bisa di-hash
        key = (args, tuple(sorted(kwargs.items())))

        # Cek apakah hasil sudah ada di cache
        if key in cached_results:
            print(f"Mengambil dari cache untuk {func.__name__}{args}")
            return cached_results[key]

        # Jika belum ada, hitung dan simpan di cache
        print(f"Menghitung {func.__name__}{args}...")
        hasil = func(*args, **kwargs)
        cached_results[key] = hasil

        return hasil

    return wrapper

@cache
def fibonacci(n):
    """Menghitung fibonacci dengan caching"""
    time.sleep(0.1)  # Simulasi komputasi berat
    if n <= 1:
        return n
    # Panggil versi yang sudah di-cache
    return fibonacci(n-1) + fibonacci(n-2)

# Test caching
print("=" * 60)
print("Pertama kali memanggil fibonacci(5):")
hasil1 = fibonacci(5)
print(f"Hasil: {hasil1}\n")
```

```
print("=" * 60)
print("Memanggil fibonacci(5) lagi (harusnya dari cache):")
hasil2 = fibonacci(5)
print(f"Hasil: {hasil2}\n")

print("=" * 60)
print("Memanggil fibonacci(6) (sebagian dari cache):")
hasil3 = fibonacci(6)
print(f"Hasil: {hasil3}")
```

```
============================================================
Pertama kali memanggil fibonacci(5):
Menghitung fibonacci(5,)...
Menghitung fibonacci(4,)...
Menghitung fibonacci(3,)...
Menghitung fibonacci(2,)...
Menghitung fibonacci(1,)...
Menghitung fibonacci(0,)...
Mengambil dari cache untuk fibonacci(1,)
Mengambil dari cache untuk fibonacci(2,)
Mengambil dari cache untuk fibonacci(3,)
Hasil: 5

============================================================
Memanggil fibonacci(5) lagi (harusnya dari cache):
Mengambil dari cache untuk fibonacci(5,)
Hasil: 5

============================================================
Memanggil fibonacci(6) (sebagian dari cache):
Menghitung fibonacci(6,)...
Mengambil dari cache untuk fibonacci(5,)
Mengambil dari cache untuk fibonacci(4,)
Hasil: 8
```

5.5 Bonus: Stacking Multiple Decorators

Kita bisa menumpuk (stack) beberapa decorator pada satu fungsi:

```
from functools import wraps
import time

# Decorator 1: Timing
def timing(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
```

```
        start = time.time()
        hasil = func(*args, **kwargs)
        end = time.time()
        print(f"⏱️  Waktu: {end - start:.4f} detik")
        return hasil
    return wrapper

# Decorator 2: Logging
def logging(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Memanggil: {func.__name__}{args}")
        hasil = func(*args, **kwargs)
        print(f"Hasil: {hasil}")
        return hasil
    return wrapper

# Decorator 3: Border
def border(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("=" * 50)
        hasil = func(*args, **kwargs)
        print("=" * 50)
        return hasil
    return wrapper

# Menumpuk decorator (dibaca dari bawah ke atas)
@border
@logging
@timing
def hitung_kuadrat(n):
    """Menghitung kuadrat dari n"""
    time.sleep(0.1)  # Simulasi delay
    return n ** 2

# Test
hasil = hitung_kuadrat(5)
```

```
==================================================
Memanggil: hitung_kuadrat(5,)
⏱️  Waktu: 0.1028 detik
Hasil: 25
==================================================
```

Urutan Eksekusi Decorator:

```
@border
@logging
@timing
def hitung_kuadrat(n):
    ...
```

Ekuivalen dengan:

```
hitung_kuadrat = border(logging(timing(hitung_kuadrat)))
```

Urutan eksekusi (dari luar ke dalam): 1. `border` wrapper - print "===" (sebelum) 2. `logging` wrapper - print "Memanggil..." 3. `timing` wrapper - catat start time 4. **Fungsi asli dieksekusi** 5. `timing` wrapper - hitung durasi 6. `logging` wrapper - print "Hasil..." 7. `border` wrapper - print "===" (setelah)

Tugas Akhir: Mini Project

Project: Sistem Validasi Input untuk Kalkulator

Buat sebuah kalkulator sederhana dengan decorator untuk: 1. **Validasi input** (harus angka) 2. **Logging** operasi 3. **Timing** eksekusi 4. **Error handling** (untuk pembagian dengan nol)

Requirements: - Buat decorator `validate_numbers` untuk memastikan semua input adalah angka - Buat decorator `log_operation` untuk mencatat operasi yang dilakukan - Buat decorator `handle_errors` untuk menangani error (khususnya division by zero) - Implementasikan fungsi: `tambah`, `kurang`, `kali`, `bagi` - Gunakan decorator stacking

Contoh output yang diharapkan:

```
[2026-05-03 10:30:45] Operasi: tambah(5, 3)
Validasi input berhasil
Hasil: 8

[2026-05-03 10:30:46] Operasi: bagi(10, 0)
Validasi input berhasil
Error: Tidak bisa membagi dengan nol!
Hasil: None
```

```
# TULIS KODE ANDA DI SINI
from functools import wraps
from datetime import datetime

# TODO: Buat decorator validate_numbers
def validate_numbers(func):
    pass

# TODO: Buat decorator log_operation
```

```python
def log_operation(func):
    pass

# TODO: Buat decorator handle_errors
def handle_errors(func):
    pass

# TODO: Implementasikan fungsi kalkulator
# Contoh:
# @handle_errors
# @log_operation
# @validate_numbers
# def tambah(a, b):
#     return a + b

# Test cases
# print(tambah(5, 3))
# print(tambah("5", 3))  # Harus error validasi
# print(bagi(10, 0))     # Harus error division by zero
```

Rangkuman

Konsep Utama yang Dipelajari:

1. **Decorator Pattern**
   - Menambahkan fungsionalitas tanpa mengubah fungsi asli
   - Menggunakan syntax `@decorator_name`
   - Memanfaatkan konsep higher-order function
2. **Closure**
   - Fungsi inner yang mengakses variabel dari outer scope
   - Dasar dari implementasi decorator
   - Memungkinkan data persistence
3. **Function Wrapping**
   - Menggunakan `@wraps` untuk mempertahankan metadata
   - Penting untuk dokumentasi dan debugging
4. **Decorator dengan Parameter**
   - Membutuhkan 3 level fungsi
   - Lebih fleksibel dan dapat dikustomisasi
5. **Aplikasi Praktis**
   - Logging: Mencatat aktivitas fungsi
   - Timing: Mengukur performa
   - Authentication: Validasi akses
   - Caching: Menyimpan hasil komputasi
   - Validation: Memvalidasi input

Referensi Tambahan

Dokumentasi Resmi:

- Python Decorators - PEP 318
- functools.wraps