Decorator

Sains Data
Functional Programming

Fungsi

Fungsi (Function)

- Menerima argumen (input) dan mengeluarkan nilai (output)
- f(x) = 2 * x
  - x = input
  - 2 * x = output

```
def greet_repeatedly(num_times):
    for i in range(num_times):
        print(“Hii”)
    return “Hi, It's nice to see you!”
```

Procedure vs Function

Procedure

Baca (x) # statement atau instruksi

var a

a = x*2 # Error

Cetak (a)

Function

Fungsi ABC(X)

    var a

    a = x*2 # Error

    return a

Cetak (Baca (x))

First-class objects

Fungsi dalam suatu bahasa pemrograman diperlakukan seperti data lainnya

Fungsi dapat:

1. Disimpan dalam variabel
2. Dikirim sebagai argumen ke fungsi lain
3. Dikembalikan sebagai nilai dari fungsi
4. Disimpan dalam struktur data (seperti list, dictionary, dsb)

First-class objects

```
# 1. Disimpan dalam variabel

def greet(name):

    return f"Hello, {name}"
```

```
# 2. Dikirim sebagai argumen

def say_something(fn, value):

    print(fn(value))

say_something(greet, "Alice")  # Output: Hello, Alice
```

First-class objects

```
# 3. Dikembalikan dari fungsi
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply

double = multiplier(2)
print(double(5))  # Output: 10
```

```
# 4. Disimpan dalam list
functions = [greet, double]
print(functions[0]("Bob"))
# Hello, Bob
print(functions )
# 6
```

Inner functions

- Fungsi di dalam fungsi lain
- Dapat diakses dalam fungsi parent

```
def outer_function():
    def inner_function_one():
        print('Calling first inner function')
    def inner_function_two():
        print('Calling second inner function')
    inner_function_one()
    inner_function_two()
outer_function()
```

Higher-order functions (HOF)

Higher-Order Functions (HOF) adalah fungsi yang bisa melakukan setidaknya satu dari dua hal berikut:

1. Menerima fungsi lain sebagai argumen.
2. Mengembalikan fungsi sebagai hasil dari eksekusinya

Pada Javascript disebut dengan “callback”

Higher-order functions (HOF)

 # HOF menerima fungsi sebagai argumen

```
def apply_operation(a, b, operation):
    return operation(a, b)
def add(x, y):
    return x + y
print(apply_operation(3, 4, add))
# Output: 7
```

 # HOF yang mengembalikan fungsi

```
def multiplier(factor):
    def multiply(x):
        return x * factor
    return multiply
double = multiplier(2)
print(double(5))
# Output: 10
```

Higher-order functions (HOF)

Build function:

- map()
- filter()
- reduce() (di functools dalam Python)
- forEach() (di JavaScript)
- sort() dengan key function

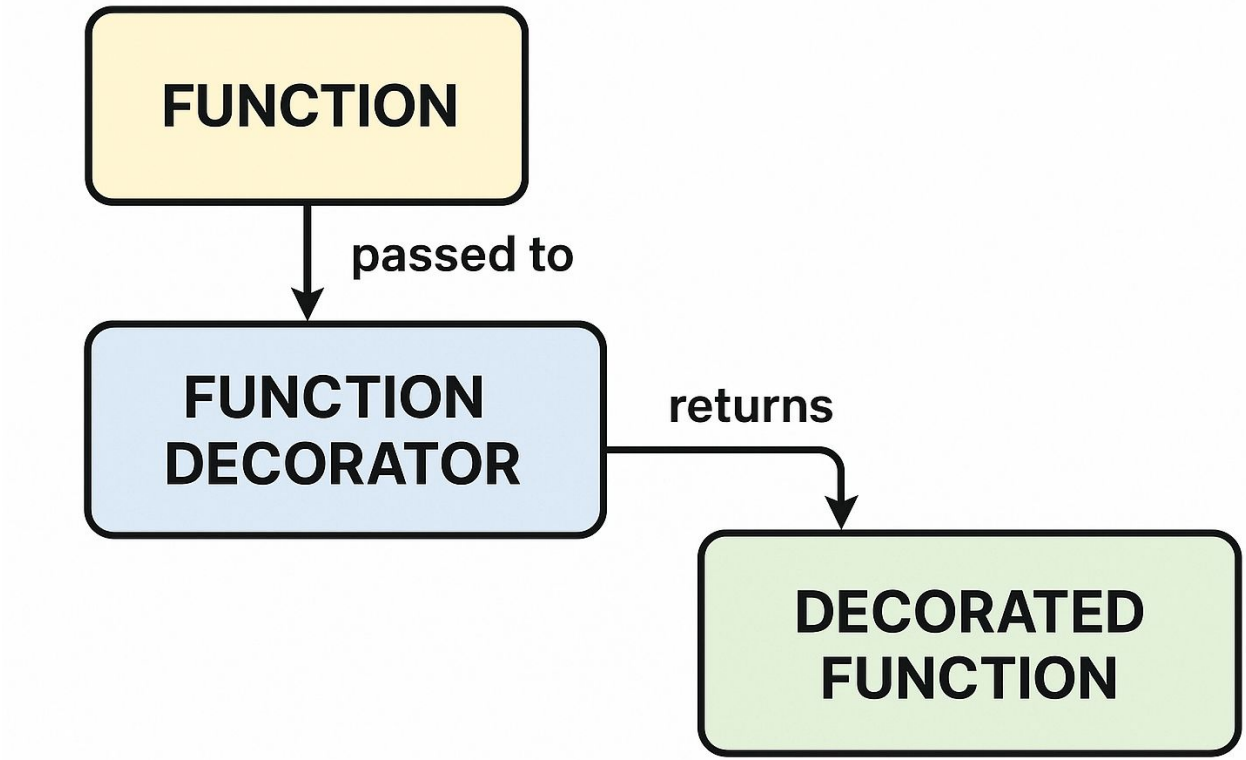
Decorator

Mengapa ?

- Jika ada beberapa fungsi yang memiliki tugas berbeda, tetapi terdapat kemiripan
- Menempatkan “sifat” kemiripan ini pada satu fungsi dan fungsi inilah yang akan mendekorasi fungsi-fungsi lain
- DRY: "Don't repeat yourself"

Decorator

- fungsi yang menerima fungsi lain sebagai argumen dan mengeluarkan fungsi tersebut dengan penambahan sifat dan fungsionalitas tertentu


FUNCTION
passed to
FUNCTION DECORATOR
returns
DECORATED FUNCTION

Contoh

Membuat decorator yang mencetak pesan sebelum dan sesudah fungsi dijalankan.

```
Sebelum halo
Halo, dunia!
Setelah halo
```

```
def say_hello():
        print("Sebelum halo")
        print("Halo, dunia!")
        print("Setelah halo")
```

```
def simple_decorator(func):
    def bungkus():
        print("Sebelum halo")
        func()
        print("Setelah halo")
    return bungkus

def say_hello():
    print("Halo, dunia!")

# Panggil fungsi
say_hello = simple_decorator(say_hello)
say_hello()
say_hello.__name__ #bungkus
```

Contoh

Membuat decorator yang mencetak pesan sebelum dan sesudah fungsi dijalankan.

```
def simple_decorator(func):
    def wrapper():
        print("Sebelum fungsi dipanggil.")
        func()
        print("Setelah fungsi dipanggil.")
    return wrapper

@simple_decorator
def say_hello():
    print("Halo, dunia!")

# Panggil fungsi
say_hello()
```

Tujuan

Untuk menambahkan fungsionalitas tambahan ke fungsi atau objek tanpa mengubah kode aslinya.

Decorator sangat berguna untuk menjaga **kode tetap bersih, modular, dan reusable.**

- Memisahkan Logika Tambahan dari Fungsi Utama
- Menghindari Duplikasi Kode
- Mendukung Open/Closed Principle
- Meningkatkan Rentabilitas
- Mempermudah Testing dan Debugging

Tujuan

Memisahkan Logika Tambahan dari Fungsi Utama

```
def log_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"[LOG] Memanggil fungsi {func.__name__} dengan argumen {args}")
        result = func(*args, **kwargs)
        print(f"[LOG] Fungsi {func.__name__} mengembalikan: {result}")
        return result
    return wrapper

@log_decorator
def tambah(a, b):
    return a + b

tambah(3, 4)
```

```
[LOG] Memanggil fungsi tambah dengan argumen (3, 4)
[LOG] Fungsi tambah mengembalikan: 7
```

Tujuan

Menghindari Duplikasi Kode (Validasi Input)

```
def validate_nonzero(func):
    def wrapper(a, b):
        if b == 0:
            return "Tidak bisa dibagi dengan nol!"
        return func(a, b)
    return wrapper
# def validate_input(func):
@validate_nonzero
def bagi(a, b):
    return a / b

print(bagi(10, 2))  # 5.0
print(bagi(10, 0))  # Tidak bisa dibagi dengan nol!
```

Tujuan

Open/Closed Principle (Tambahkan fitur tanpa ubah fungsi)

```
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

@uppercase_decorator
def greet(name):
    return f"Halo, {name}"

print(greet("dunia"))  # HALO, DUNIA
```

Tujuan

Reusability (Decorator digunakan di banyak fungsi)

```
def simple_log(func):
    def wrapper(*args, **kwargs):
        print(f"[LOG] Memanggil {func.__name__}")
        return func(*args, **kwargs)
    return wrapper
@simple_log
def salam():
    print("Halo!")
@simple_log
def pamit():
    print("Sampai jumpa!")

salam()
pamit()
```

```
[LOG] Memanggil salam
Halo!
[LOG] Memanggil pamit
Sampai jumpa!
```

Tujuan

Testing/Maintainability

```
def debug_decorator(func):
    def wrapper(*args, **kwargs):
        print(f"[DEBUG] Args: {args}, Kwargs: {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@debug_decorator
def perkalian(a, b):
    return a * b

perkalian(5, 3)
```

```
[DEBUG] Args: (5, 3), Kwargs: {}
15
```

Soal 1

Buatlah sebuah decorator bernama log_decorator yang:

- mencetak pesan sebelum dan sesudah fungsi dipanggil.
- diterapkan pada fungsi hello(name) yang mencetak "Hello, <name>".

Output
Memulai fungsi...
Hello, Budi
Fungsi selesai.

Soal 2

Buat decorator double_result yang:

- Mengalikan hasil return dari fungsi dengan 2

```
@double_result
def get_number():
    return 5

# output
10
```

Soal 3

Buat decorator log_args yang mencetak semua argumen yang dikirim ke fungsi.

```
@log_args
def multiply(a, b):
    return a * b

multiply(3, 4)

#output
Fungsi dipanggil dengan argumen: (3, 4)
```

Soal 4

Buatlah decorator bernama repeat(n) yang menjalankan fungsi sebanyak n kali.

```
@repeat(n)
def say_hi():
    print("Hi!")

Say_hi (3)
# Output
Hi!
Hi!
Hi!
```

Praktikum Decorator

Sains Data

Apa itu decorator

- Decorator adalah fitur Python yang memungkinkan Anda memodifikasi perilaku fungsi atau metode tanpa mengubah kode aslinya.
- Decorator bekerja dengan membungkus fungsi target menggunakan fungsi lain, sehingga Anda dapat menambahkan logika tambahan sebelum atau sesudah fungsi tersebut dijalankan.
- Fitur ini sangat berguna untuk logging, validasi, kontrol akses, caching, dan lainny

Cara Kerja Decorator

- Decorator memanfaatkan konsep fungsi tingkat tinggi (higher-order functions), yaitu fungsi yang dapat menerima fungsi lain sebagai argumen dan/atau mengembalikan fungsi.
- Dalam Python, fungsi adalah objek tingkat pertama (first-class objects), artinya mereka dapat diperlakukan seperti data biasa—dikirim sebagai argumen, disimpan dalam variabel, atau dikembalikan dari fungsi lain.

Contoh decorator

```python
def my_decorator(func):
    def wrapper():
        print("Sebelum fungsi dijalankan.")
        func()
        print("Setelah fungsi dijalankan.")
    return wrapper

def say_hello():
    print("Hello!")

```

```python
say_hello = my_decorator(say_hello)
say_hello()
```

```
Sebelum fungsi dijalankan.
Hello!
Setelah fungsi dijalankan.
```

- **say_hello = my_decorator(say_hello)**.
- higher-order functions
- say_hello()

Contoh decorator

```python
# OR
def my_decorator(func):
    def wrapper():
        print("Sebelum fungsi dijalankan.")
        func()
        print("Setelah fungsi dijalankan.")
    return wrapper
@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
Sebelum fungsi dijalankan.
Hello!
Setelah fungsi dijalankan.
```

- Dalam contoh, **@my_decorator** setara dengan **say_hello = my_decorator(say_hello)**.
- Ketika **say_hello**() dipanggil, sebenarnya yang dijalankan adalah fungsi wrapper yang membungkus **say_hello**

Decorator dengan argument

- Dekorator dengan menerima argumen.
- Ini memungkinkan fungsi luar menerima argumen dan mengembalikan fungsi dekorator yang sebenarnya.
- **repeat_decorator** dengan argumen times dan mengembalikan fungsi decorator yang sebenarnya. Fungsi decorator mendefinisikan fungsi wrapper, yang mengulang panggilan ke fungsi asli times.

```python
# Decorators with Arguments
def repeat_decorator(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat_decorator(3)
def greet(name):
    print(f"Hello, {name}!")

greet("Kaitlyn")
```

The *functools.wraps* Decorator

- Gunakan functools.wraps untuk menyimpan metadata (seperti nama dan docstring) dari fungsi asli.
- Tanpa functools.wraps, metadata dari fungsi asli akan hilang, dan fungsi yang didekorasi akan berakhir dengan metadata dari fungsi wrapper.

The *functools.wraps* Decorator

```python
# tidak Mennggunakan  wraps
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Wrapper called")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def example():
    """This is an example function"""
    print("Example function")

print(example.__name__)  # Output: wrapper
print(example.__doc__)   # Output: None
```

```python
# Mennggunakan  wraps
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Wrapper called")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def example():
    """This is an example function"""
    print("Example function")

print(example.__name__)  # Output: example
print(example.__doc__)   # Output: This is an example function
```

Apa tujuannya metadata dari fungsi asli (example) harus dipertahankan (tidak hilang) ?

Chaining Decorators

- Menerapkan beberapa dekorator ke satu fungsi (chaining decorators)
- Setiap dekorator diterapkan dalam urutan yang tercantum

```python
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

def exclamation_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result + "!"
    return wrapper

@uppercase_decorator
@exclamation_decorator
def greet(name):
    return f"Hello, {name}"

print(greet("Kaitlyn"))  # Output: HELLO, KAITLYN!
```

Task di praktikum

Bahas Soal No 1, 2, 3, dan 4