Unraveling the Magic of Closures: A Deep Dive into Syntactic Refreshment, Decorators, and Memoization

Dearni Monica Manik, Nabila Anilda Zahrah, Abit Ahmad Oktarian,
Allya Nurul Islami Pasha, Yohana Manik, David Boby C.Nainggolan

Program Studi Sains Data Institut Teknologi Sumatera

Jl. Terusan Ryacudu, Way Huwi, Kec. Jatiagung, Kabupaten Lampung Selatan,

Lampung 35365

Email:
nabila.122450063@student.itera.ac.id dearni.122450075@student.itera.ac.id
abit.122450042@student.itera.ac.id allya.122450033@student.itera.ac.id
david.122450048@student.itera.ac.id yohana.122450126@student.itera.ac.id

Pendahuluan

Dalam dunia pemrograman, menciptakan kode yang efisien, mudah dipahami, dan elegan adalah tujuan utama setiap pengembang. Untuk mencapai hal ini, penting untuk memahami konsep-konsep seperti closure, decorator, dan memoization, yang dapat membantu meningkatkan kualitas dan kinerja kode.

Dalam artikel ini, kami akan memaparkan implementasi closure dengan syntactic sugar dengan konsep syntactic sugar, decorator dan memorization. Kami akan membahas cara-cara baru untuk menerapkan closure dengan sintaks yang lebih bersih dan intuitif, bagaimana menggunakan decorator untuk menambahkan fungsionalitas tambahan dengan mudah, serta bagaimana memanfaatkan memoization untuk meningkatkan efisiensi komputasi.

Dengan artikel ini diharapkan dapat memberi pemahaman yang mendalam tentang konsep-konsep ini dan kemampuan untuk menerapkannya dengan cara yang inovatif, sehingga dapat menghasilkan kode yang lebih baik, lebih cepat, dan lebih mudah dipahami.

Metode

Dalam membuat sebuah program akan digunakan metode metode pendukungnya. Dalam analisis ini digunakan tiga konsep fundamental dalam pemrograman fungsional dan pemrograman berorientasi objek. Setiap konsep dalam pemrograman memiliki peran penting dalam meningkatkan kejelasan, fleksibilitas, dan efisiensi kode. Metode yang digunakan dalam analisis ini, yaitu :

1. Closures

   Metode closure adalah konsep dalam pemrograman Python yang digunakan untuk menyimpan fungsi dalam variabel. Fungsi closure dapat disimpan ke variabel dan

kemudian dieksekusi ketika diperlukan. Metode closure banyak digunakan pada operasi data sequence atau agregasi data numerik.

2. Decorators

Dekorator adalah fungsi dalam pemrograman yang mengubah perilaku fungsi atau metode lain tanpa mengubah kode sumber aslinya. Decorators berfungsi sebagai lapisan tambahan yang membungkus fungsi yang ada, yang memungkinkan pengguna untuk menambahkan fungsionalitas tambahan sebelum atau setelah fungsi dijalankan. Dekorator sangat berguna dalam mengatur dan mengelola kode, terutama ketika pengguna ingin menambahkan fungsionalitas tertentu ke beberapa bagian kode Anda.

Prinsip dasar dalam penggunaan decorators :

- Create a Decorator : Dalam menggunakan decorator pengguna harus membuat fungsi decorator terlebih dahulu. Fungsi ini biasanya menggunakan fungsi lain sebagai argumen dan mengembalikan fungsi baru yang diperluas. Pengguna dapat menerapkan dekorator ke fungsi yang ingin diubah dengan menggunakan simbol "@" sebelum definisi fungsi.
- Using a Decorator : Pengguna dapat menerapkan decorator ke fungsi yang ingin di modifikasi dengan menggunakan simbol "@" sebelum definisi fungsi.

Dekorator dengan Python penting untuk meningkatkan keterbacaan dan pemeliharaan kode, memungkinkan pengguna memisahkan tugas tertentu ke dalam lapisan terpisah. Tugas seperti validasi input, pencatatan, pengukuran waktu eksekusi, pengelolaan izin atau autentikasi, dan hasil fungsi pemrosesan adalah contoh tugas yang menggunakannya.

3. Memoization

Metode pemrograman yang dikenal sebagai memoization adalah penggunaan cache untuk menyimpan hasil komputasi. Memoization adalah teknik yang digunakan dalam pemrograman untuk mempercepat operasi rekursif berulang yang sering terjadi dan bertujuan untuk mengurangi jumlah waktu yang diperlukan untuk mengulangi proses komputasi yang sama.

Dalam Penerapan memoization dapat menggunakan fungsi bantu yang menyimpan hasil komputasi yang sudah dilakukan sebelumnya. Fungsi bantu ini akan mengembalikan hasil yang sudah dihitung sebelumnya jika komputasi yang sama terjadi kembali, sehingga tidak perlu melakukan komputasi lagi. Hal ini mempercepat kinerja program karena tidak perlu melakukan komputasi yang sama berulang kali.

Pengulangan fungsi dengan argumen yang sama atau pengulangan fungsi dengan banyak komputasi adalah dua contoh situasi di mana memori dapat digunakan. Memoisasi dapat mempercepat kinerja dan mempersingkat waktu proses.

Pembahasan

Berikut pembahasan kami mengenai penggunaan decorator, memoization, dan syntactic sugar pada python:

```python
# Implementasi Closures dengan Syntactic Sugar
def outer(pesan):
    def inner():
        print(pesan)
    return inner

sapa = outer("Sakti kan ini closure")
sapa()  # Output: Sakti kan ini closure
```

Gambar 1

Kode pada Gambar 1 merupakan penerapan closures dalam Python dengan menggunakan syntactic sugar. Fungsi ‘outer’ akan mengambil satu parameter ‘pesan’ dan mengembalikan sebuah fungsi dalam lingkupnya, yaitu fungsi ‘inner’. Fungsi ‘inner’ disini tidak memiliki parameter, namun memiliki akses ke variabel ‘pesan’ yang didefinisikan di lingkup luar. Ketika fungsi ‘outer’ dipanggil dengan argumen “Sakti kan ini closure”, itu menghasilkan sebuah fungsi yang kemudian disimpan dalam variabel ‘sapa’. Kemudian, saat fungsi ‘sapa’ dipanggil, ia akan mencetak nilai ‘pesan’ yang ditentukan saat fungsi ‘outer’ pertama kali dipanggil, yaitu “sakti kan ini closure”. Sehingga kode tersebut akan menghasilkan output dari pemanggilan ‘sapa( )’ adalah “sakti kan ini closure”. Untuk outputnya dapat dilihat pada Gambar 2 dibawah ini.

```python
# Implementasi Closures dengan Syntactic Sugar
def outer(pesan):
    def inner():
        print(pesan)
    return inner

sapa = outer("Sakti kan ini closure")
sapa()  # Output: Sakti kan ini closure
```

```
Sakti kan ini closure
```

Gambar 2

```python
# Implementasi Decorator
def timing_decorator(func):
    import time

    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"waktu eksekusi untuk {func.__name__}: {end_time - start_time} detik")
        return result

    return wrapper

@timing_decorator
def komputasi_brutal(n):
    # Misalkan ini adalah komputasi berat yang memakan waktu
    result = sum(range(n))
    return result

print(komputasi_brutal(10000))
```

Gambar 3

Pada Gambar 3 menunjukkan penggunaan decorator di Python. Fungsi ‘timing_decorator’ adalah sebuah decorator yang menerima fungsi lain (‘func’) sebagai argumen. Decorator ini menambahkan fungsionalitas untuk mengukur waktu eksekusi dari fungsi yang didekorasi. Fungsi ‘wrapper’ dalam decorator ini mengukur waktu mulai sebelum fungsi yang didekorasi dijalankan, kemudian menjalankan fungsi tersebut dengan argumen yang sesuai, dan akan menghitung waktu selesai setelah fungsi selesai dieksekusi. Informasi waktu eksekusi kemudian dicetak ke layar bersama dengan nama fungsi yang dieksekusi. Fungsi ‘wrapper’ ini kemudian dikembalikan oleh decorator untuk menggantikan fungsi asli yang didekorasi. Fungsi ‘komputasi_brutal’ adalah fungsi yang didekorasi dengan decorator ‘timing_decorator’. Ketika fungsi ‘komputasi_brutal’ dipanggil dengan argumen ‘10000’, decorator akan mengukur waktu eksekusi fungsi tersebut dan mencetak informasi tersebut ke layar seperti pada Gambar 4 dibawah, bersama dengan hasil komputasi yang dikembalikan oleh fungsi tersebut. Waktu komputasi nya adalah 0.00020766258239746094.

```python
# Implementasi Decorator
def timing_decorator(func):
    import time

    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"waktu eksekusi untuk {func.__name__}: {end_time - start_time} detik")
        return result

    return wrapper

@timing_decorator
def komputasi_brutal(n):
    # Misalkan ini adalah komputasi berat yang memakan waktu
    result = sum(range(n))
    return result

print(komputasi_brutal(10000))
```

```
waktu eksekusi untuk komputasi_brutal: 0.00020766258239746094 detik
49995000
```

Gambar 4

```python
# Implementasi Memoization
def memoize(func):
    cache = {}

    def memoized_func(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]

    return memoized_func

@memoize
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

Gambar 5

Pada Gambar 5 adalah sebuah bentuk implementasi memoization dalam Python. Fungsi `memoize` adalah sebuah decorator yang digunakan untuk menyimpan hasil pemanggilan fungsi `fibonacci` dalam sebuah cache untuk mempercepat pemanggilan berikutnya dengan argumen yang sama. Saat fungsi `fibonacci` dipanggil dengan argumen `n`, decorator akan memeriksa apakah nilai `n` sudah ada dalam cache. Jika sudah ada, maka nilai tersebut langsung diambil dari cache tanpa perlu menghitung ulang. Jika belum ada, fungsi `fibonacci` akan dipanggil secara rekursif untuk menghitung nilai `n`, dan hasilnya akan disimpan di dalam cache untuk pemanggilan berikutnya. Dengan demikian, ketika fungsi `fibonacci` dipanggil dengan argumen yang sama, hasilnya akan langsung diambil dari cache tanpa perlu melakukan perhitungan ulang. Ini dapat menghemat waktu komputasi, terutama untuk kasus di mana fungsi tersebut sering dipanggil dengan argumen yang sama. Pada gambar 5 ini, saat fungsi `fibonacci` dipanggil dengan argumen `10`, hasilnya akan dihitung dan dicetak. Output dari kode pada Gambar 5 adalah 55, dapat dilihat pada Gambar 6 dibawah. Penggunaan memoization ini dapat membantu dalam menghitung bilangan fibonacci lebih efisien.

```python
# Implementasi Memoization
def memoize(func):
    cache = {}

    def memoized_func(n):
        if n not in cache:
            cache[n] = func(n)
        return cache[n]

    return memoized_func

@memoize
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

```
55
```

Gambar 6

Kesimpulan

Berdasarkan penerapan closure, dekorator, dan memoization yang telah dilakukan didapatkan kesimpulan sebagai berikut:

1. Closure memungkinkan fungsi inner untuk mengakses variabel di lingkup luar, sehingga memungkinkan fleksibilitas dalam penggunaannya.
2. Penggunaan decorator dapat menambahkan fungsionalitas tanpa mengubah kode aslinya.
3. Simbol @ kemudian diikuti nama fungsi menunjukkan penerapan decorator dan fungsi yang akan dipakai.
4. Memoization mempercepat waktu eksekusi dengan menyimpan hasil perhitungan sebelumnya dalam cache, seperti dalam kode yang dilakukan yaitu menghitung bilangan fibonacci
5. Memoization yang didapat dalam penerapan pada kode adalah 0.00020766258239746094.