Modul 5 Praktikum Pemrograman Berbasis Fungsi

March 11, 2023

1 Modul 5 Praktikum Pemrograman Berbasis Fungsi

Program Studi Sains Data

Institut Teknologi Sumatera

Tujuan 1. Mahasiswa diharapkan dapat menerapkan konsep python scope dan closure 2. Mahasiswa diharapkan dapat membuat pemrograman sederhana dengan konsep closure

1.1 Pemrograman Sederhana dengan Closure

Jika kita diberikan sebuah fungsi yang kemudian di simpan dalam sebuah list. Lalu print hasilnya mengapa keluarannya berupa angka yang sama?

[ ]:
```
def funcs_generator():
    funcs = []
    for i in range(3):
        def f():
            return i * 2
        funcs.append(f)
    return funcs

f1, f2, f3 = funcs_generator()
print(f1(), f2(), f3())
```

```
4 4 4
```

Review lah kode di atas, kita dapat melihat enclosed variabel i telah sama dengan 2 ketika kita mengeksekusi f1(), f2(), f3() dalam fungsi print(). jika kita print `__closure__` dari fungsi ini. maka hasilnya adalah

[ ]:
```
def funcs_generator():
    funcs = []
    for i in range(3):
        def f():
            return i * 2
        funcs.append(f)
    return funcs

f1, f2, f3 = funcs_generator()
```

```
print(f1(), f2(), f3())
print(f1.__closure__[0].cell_contents)
print(f2.__closure__[0].cell_contents)
print(f3.__closure__[0].cell_contents)
```

```
4 4 4
2
2
2
```

Dari pemrograman di atas, kita pasti bertanya bagaimana mendapatkan output dengan keluaran yang berbeda? Bagaimana caranya?

[ ]:
```
def funcs_generator():
    funcs = []
    for i in range(3):
        def f(j = i):
            return j * 2
        funcs.append(f)
    return funcs

f1, f2, f3 = funcs_generator()
print(f1(), f2(), f3())
print(f1.__closure__)
print(f2.__closure__)
print(f3.__closure__)
```

```
0 2 4
None
None
None
```

Seperti yang ditunjukkan di atas, kita menggunakan variabel j untuk menerima parameter i, dan hasilnya seperti yang diharapkan. Namun, kali ini ketiga fungsi tersebut bukan merupakan closure karena tidak perlu "mengingat" variabel apa pun dalam enclosed scope.

Ingat kondisi kedua dari 3 kondisi untuk membangun closure yang disebutkan di modul sebelumnya: *"Inner Function harus menggunakan variabel yang didefinisikan dalam fungsi luarnya."*

pada contoh di atas, variabel i tidak digunakan dalam inner function. yang mana menggunakan nilai sebagai argumen ke parameter j di inner function. Meskipun, closure tidak dibuat.

Kita harus berhati-hati ketika inner function sebagai referensi beberapa enclosing variabel yang mana akan berubah setelahnya. Berikut contoh closure menggunakan lambda

[ ]:
```
def outer_func():
    leader = "Hi Sains Data"
    return lambda _: print(leader)

f = outer_func()
print(outer_func.__closure__)
```

```
print(f.__closure__)
print(f.__closure__[0].cell_contents)
```

```
None
(<cell at 0x000001591923A7A0: str object at 0x0000015919216B70>,)
Hi Sains Data
```

1.1.1 Fungsi Komposisi

Jika kita memiliki dua fungsi f dan g, dan kita ingin mengkombinasi fungsi ini yang mana output dari f akan menjadi input untuk g. Proses ini disebut fungsi komposisi. Jadi operasi komposisi mengambil 2 fungsi f dan g yang menghasilkan fungsi h, yaitu h(x)=g(f(x)). Kita dapat mengimplementasikannya dalam bentuk closure:

[ ]:
```
def compose(g, f):
    def h(*args, **kwargs):
        return g(f(*args, **kwargs))
    return h
```

h dalam closure yang digunakan menyimpan nilai non local variabel f dan g. variabel nonlocal ini adalah fungsi dari mereka sendiri. dimana kita menggunakan *args dan **kwargs untuk melewati multiple argument atau argumen yang merupakan katakunci untuk h. Berikut contoh applikasi sederhana yaitu konversi inch ke foot dan foot ke meter.

[ ]:
```
inch_to_foot= lambda x: x/12
foot_meter= lambda x: x * 0.3048
inch_to_meter = compose(foot_meter, inch_to_foot)
inch_to_meter(12)
```

[ ]:
```
0.3048
```

1.1.2 Partial Application

Dalam matematika, jumlah argumen yang dibutuhkan suatu fungsi disebut arity dari fungsi tersebut. Aplikasi parsial adalah operasi yang mengurangi aritas suatu fungsi. yang berarti memungkinkan Anda untuk memperbaiki nilai beberapa argumen dan membekukannya untuk mendapatkan fungsi dengan parameter yang lebih sedikit.

Sebagai contoh, arity dari f(x,y,z) adalah 3. kita dapat memperbaiki nilai dari argumen x untuk memiliki f(x=a, y,z) = g(y,z). maka arity dari g(y,z) adalah 2, dan memiliki hasil partial application dari f(x,y,z). jadi parsial(f) => g. berikut aplikasi parsial menggunakan closures:

[ ]:
```
def partial(f, *f_args, **f_keywords):
    def g(*args, **keywords):
        new_keywords = f_keywords.copy()
        new_keywords.update(keywords)
        return f(*(f_args + args), **new_keywords)
    return g
```

Disini outer function menerima nilai f dan posisi dan katakunci argumen dari f bahwa perlu dilakukan perbaikan. inner function g menambahkan argumen ini ke argumen yang tersisa dari f yang diterima nanti sebagai fungsi parsial. Akhirnya, memanggil f dengan semua argumen yang sudah terkumpul. Berikut contoh nya:

[ ]:
```
func = lambda x,y,z: x**2 + 2*y + z
pfunc = partial(func, 1)
pfunc(2, 3)  # Output is 8
```

[ ]:
```
8
```

2 Penerapan Closure pada Python Decorator

Decorator adalah fungsi yang mengambil pemanggilan lain seperti fungsi, metode, dan class yang mana tidak memerlukan modifikasi secara eksplisit.

[ ]:
```
def decorator(func):
    def wrapper():
      print("Before the function gets called.")
      func()
      print("After the function is executed.")
    return wrapper
def wrap_me():
    print("Hello Sains Data!")
wrap_me = decorator(wrap_me)
wrap_me()
```

```
Before the function gets called.
Hello Sains Data!
After the function is executed.
```

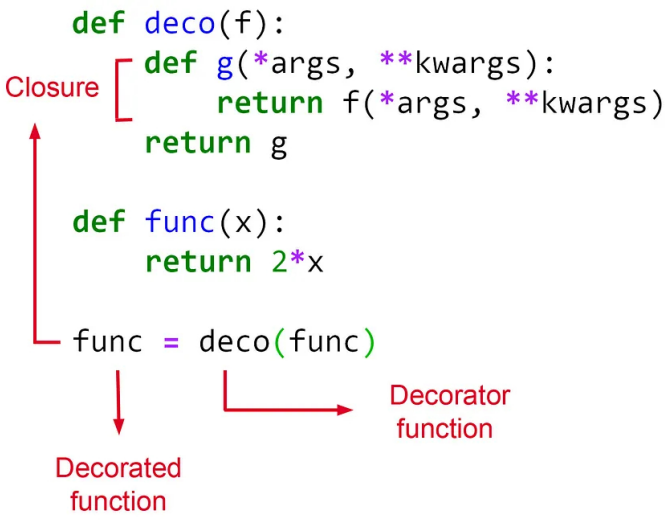
Kita memanggil decorator dengan melewati fungsi wrap_me, ketika decorator mendapat panggilan. Wrapper function didefinisikan untuk menahan referensi dari wrap_me (yaitu closure) dan digunakan untuk membungkus input function (wrap_me) untuk mengubah perilakunya pada waktu pemanggilan.

Di sini fungsi luar atau outer function adalah deco(f) yang mengambil fungsi f sebagai argumen, dan fungsi dalam g didefinisikan sebagai closure. Kita mendefinisikan fungsi lain func dan menerapkan deco padanya. Tetapi untuk menginisialisasi closure kita menetapkan hasil deco ke func. Jadi deco mengambil func sebagai argumen dan menugaskan penutupannya ke func lagi. Dalam hal ini, kita mengatakan bahwa func didekorasi dengan deco dan deco adalah dekorator untuk func.

[ ]:
```
def deco(f):
    def g(*args, **kwargs):
        return f(*args, **kwargs)
    return g
def func(x):
     return 2*x
func = deco(func)
```

[ ]:
```
from IPython import display
display.Image(r"C:\Users\ardik\OneDrive\Pythonenv\Praktikum PBF\func1.jpg")
```

[ ]:



Penting untuk diperhatikan bahwa setelah menugaskan hasil dekorator ke func, func mengacu pada closure g. Jadi memanggil func(a) seperti memanggil g(a). maka:

[ ]:
```
func.__name__
```

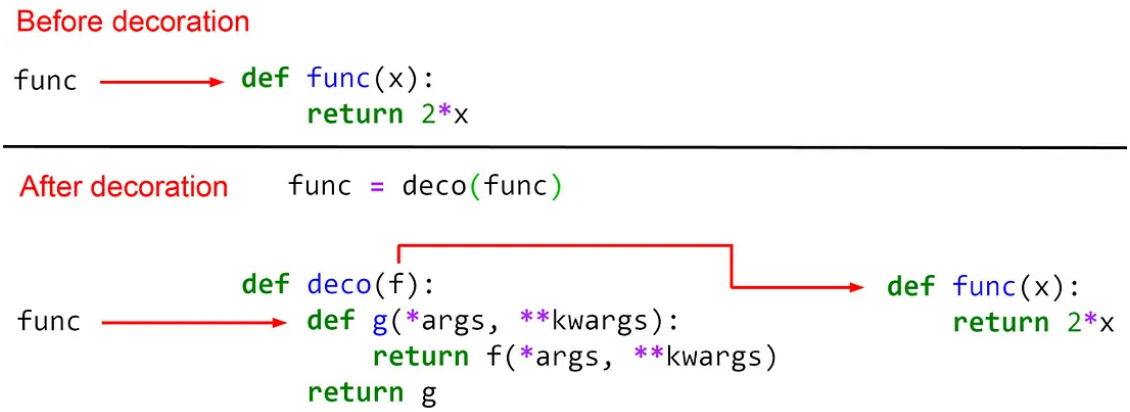
[ ]:
```
'g'
```

hasilnya akan menjadi 'g'. Faktanya, variabel func hanyalah referensi ke definisi fungsi. Awalnya, ini merujuk ke definisi func(x), tetapi setelah dekorasi, ini merujuk ke g(*args, **kwargs).

Tapi apa yang terjadi pada fungsi asli func(x)? Apakah kita akan kehilangannya? Ingatlah bahwa dekorator menerima fungsi sebagai argumennya. Jadi fungsi ini memiliki salinan referensi ke func sebagai variabel lokalnya f. Jika referensi asli berubah, itu tidak mempengaruhi variabel lokal tersebut, jadi di dalam g, f masih mengacu pada definisi func(x) (menurut definisi fungsi, fungsi func(x) seperti yang ditunjukkan pada Gambar di bawah ini).

[ ]:
```
from IPython import display
display.Image(r"C:\Users\ardik\OneDrive\Pythonenv\Praktikum PBF\func2.jpg")
```

[ ]:



Jadi untuk meringkasnya, setelah dekorasi, variabel func merujuk ke closure g, dan di dalam g, f merujuk ke definisi func(x). Nyatanya g sekarang bertindak sebagai antarmuka untuk fungsi asli func(x) yang didekorasi. Kita tidak dapat langsung memanggil fungsi (x) di luar g. Sebaliknya, pertama-tama kita memanggil func untuk memanggil g, dan kemudian di dalam g kita dapat memanggil f untuk memanggil fungsi asli func(x). Jadi kita memanggil fungsi asli func(x) menggunakan closure g.

Sekarang dengan menggunakan closure ini, kita dapat menambahkan beberapa kode lagi untuk dijalankan sebelum atau sesudah memanggil func(x). Misalkan kita ingin men-debug kode kita sebelumnya. Kita ingin tahu kapan func(x) dipanggil. maka cukup menempatkan pernyataan print di dalam func(x), untuk memberi tahu kita saat dijalankan:

[ ]:
```
def func(x):
    print("func is called")
    return 2*x
```

Tapi hal ini malah mengubah fungsinya, dan kita harus ingat untuk menghapusnya nanti. Solusi yang lebih baik adalah mendefinisikan dekorator untuk membungkus fungsi dan menambahkan pernyataan print ke closure itu.

[ ]:
```
def deco(f):
    def g(*args, **kwargs):
        print("Calling ", f.__name__)
        return f(*args, **kwargs)
    return g
def func(x):
    return 2*x
func = deco(func)
func(2)
```

```
Calling  func
```

[ ]:
```
4
```

2.1 Memoization

Memoisasi adalah teknik pemrograman yang digunakan untuk mempercepat program. Berasal dari kata Latin memorandum yang berarti "untuk diingat". Seperti namanya, ini didasarkan pada memorizing atau menyimpan hasil pemanggilan fungsi yang besar. Jika input yang sama atau pemanggilan fungsi dengan parameter yang sama digunakan, hasil cache sebelumnya akan digunakan untuk menghindari perhitungan yang tidak perlu. Dengan Python, kita dapat secara otomatis membuat memo fungsi menggunakan closure dan dekorator.

Berikut fungsi yang menghitung angka Fibonacci. Angka Fibonacci didefinisikan secara rekursif. Setiap angka adalah jumlah dari dua angka sebelumnya, mulai dari 0 dan 1:

[ ]:
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
for i in range(6):
    print(fib(i), end=" ")
```

```
0 1 1 2 3 5
```

Sekarang kita mendefinisikan fungsi baru yang dapat memoize fungsi lain:

[ ]:
```
def memoize(f):
    memo = {}
    def memoized_func(n):
        if n not in memo:
            memo[n] = f(n)
        return memo[n]
    return memoized_func
```

Untuk menggunakan fungsi memoize, kita bisa menggunakannya sebagai dekorator untuk fib :

[ ]:
```
fib = memoize(fib)
fib(30)
```

[ ]:
```
832040
```

Jadi fungsi fib sekarang didekorasi dengan memoize(). fib mengacu pada closure memoized_func, jadi ketika kita memanggil fib(30), itu seperti memanggil memoized_func(30). Dekorator menerima fib asli sebagai argumennya f, jadi f mengacu pada definisi fib(n) di dalam memoized_func. Closure memoized_func pertama-tama memeriksa apakah n ada di kamus memo atau tidak. Jika ada di dalamnya, maka hanya mengembalikan memo[n], dan tidak memanggil fib(n) asli. jika n tidak ada dalam kamus memo, maka pertama kali akan memanggil f(n) yang mengacu pada fib(n) asli. Selanjutnya kemudian menyimpan hasilnya di kamus memo, dan akhirnya mengembalikannya sebagai hasil akhir.

2.2 Tracing recursive functions

Misalkan kita ingin men-debug fungsi Fibonacci rekursif yang kita definisikan. Kita ingin melihat bagaimana fungsi ini memanggil dirinya sendiri untuk menghitung hasil akhir. Kita dapat menentukan dekorator untuk melacak pemanggilan fungsi:

[ ]:
```
def trace(f):
    level = 1
    def helper(*arg):
        nonlocal level
        print((level-1)*"   ", "   ", f.__name__,
              "(", ",".join(map(str, arg)), ")", sep="")
        level += 1
        result = f(*arg)
        level -= 1
        print((level-1)*"   ", "   ", result, sep="")
        return result
    return helper
```

[ ]:
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
fib = trace(fib)
fib(4)
```

```
fib(4)
   fib(3)
      fib(2)
         fib(1)
         1
         fib(0)
         0
      1
      fib(1)
      1
   2
   fib(2)
      fib(1)
      1
      fib(0)
      0
   1
3
```

[ ]: 3

Jejak fungsi dekorator menerima fib sebagai argumennya. Setelah memanggil dekorator, fib merujuk ke pembantu closure. trace memiliki variabel nonlokal yang disebut level yang menunjukkan kedalaman rekursi dan awalnya disetel ke 1. helper pertama-tama mencetak beberapa karakter padding termasuk , , dan untuk menampilkan struktur pohon rekursi. Jumlah karakter sebanding dengan nilai level.

Saat kedalaman rekursi meningkat, lebih banyak dicetak. Untuk panggilan pertama | tidak dicetak karena level-1 sama dengan nol. Kemudian nama fungsi dan nilai parameternya saat ini dicetak. Setelah itu level dinaikkan satu untuk menunjukkan level rekursi berikutnya. Kemudian fungsi f (yang mengacu pada fungsi asli yang diteruskan ke dekorator) dievaluasi. Jadi definisi fib(n) dievaluasi. Karena fib didefinisikan secara rekursif, ia akan memanggil dirinya sendiri di beberapa titik. Ingatlah bahwa setelah dekorasi, fib mengacu pada closure helper. Jadi panggilan rekursif akan memanggil helper lagi, dan level rekursi selanjutnya akan dicetak.

Rekursi berlanjut hingga kita mencapai kasus dasar rekursi dan definisi fungsi dari fib(n) mengembalikan 1. Kemudian level dikurangi satu untuk menghubungkan hasil yang dikembalikan dengan nama fungsi yang mengembalikannya. Nilai yang dikembalikan dicetak satu per satu sampai kita kembali ke panggilan pertama.

2.3 Syntatic Sugar

Sugar sintaksis adalah sintaks dalam bahasa pemrograman yang membuat sesuatu lebih mudah dibaca atau diekspresikan. Menggunakan sintaksis ini, Anda dapat menulis kode lebih jelas atau lebih ringkas. Misalnya, dalam Python, operator += adalah sugar sintaksis. Jadi, alih-alih menulis a=a+1, Anda cukup menulis a+=1.

kita menulis `func = deco(func)` untuk menghias fungsi func. Sebenarnya kita bisa melakukannya dengan cara yang lebih sederhana, dan hasil akhirnya sama:

[ ]:
```
def deco(f):
    def g(*args, **kwargs):
        print("Calling ", f.__name__)
        return f(*args, **kwargs)
    return g
@deco
def func(x):
    return 2*x
func(2)
```

```
Calling  func
```

[ ]: 4

Sebelumnya kita menulis `func = deco(func)` lalu berikutnya kita cukup menulis @deco di atas fungsi yang seharusnya didekorasi oleh `deco(f)`. Ini adalah sugar sintaksis untuk menghias suatu fungsi dan terkadang disebut sintaks pai.

[ ]:
```
display.Image(r"C:\Users\ardik\OneDrive\Pythonenv\Praktikum PBF\func3.jpg")
```

[ ]:

```
def deco(f):
    :
    def g(*args, **kwargs):
        :
    return g
```

```
@deco
def func(x):
    :
```

=

```
def func(x):
    :
func = deco(func)
```

Beberapa catatan tentang sintaks ini. `@deco` tidak boleh berada di tempat lain dan tidak ada lagi pernyataan yang diizinkan antara `@deco` dan definisi `func(x)`. Anda tidak boleh menulis parameter deco setelah @deco. Sintaks mengasumsikan deco hanya mengambil satu parameter yang merupakan fungsi yang harus didekorasi. Nanti saya akan menjelaskan apa yang harus dilakukan jika fungsi dekorator memiliki lebih banyak parameter.

dekorasi dimulai setelah Anda menulis func = deco(func). Tapi kapan itu terjadi di contoh sebelumnya. Fitur utama dari dekorator yang didefinisikan dengan sintaks pai adalah dekorator berjalan tepat setelah fungsi yang didekorasi ditentukan. Itu biasanya pada waktu impor ketika modul dimuat oleh Python.

2.4 Stacked Decorator

[ ]:
```
def deco1(f):
    def g1(*args, **kwargs):
        print("Calling ", f.__name__, "using deco1")
        return f(*args, **kwargs)
    return g1
def deco2(f):
    def g2(*args, **kwargs):
        print("Calling ", f.__name__, "using deco2")
        return f(*args, **kwargs)
    return g2
def func(x):
    return 2*x
func = deco2(deco1(func))
func(2)
```

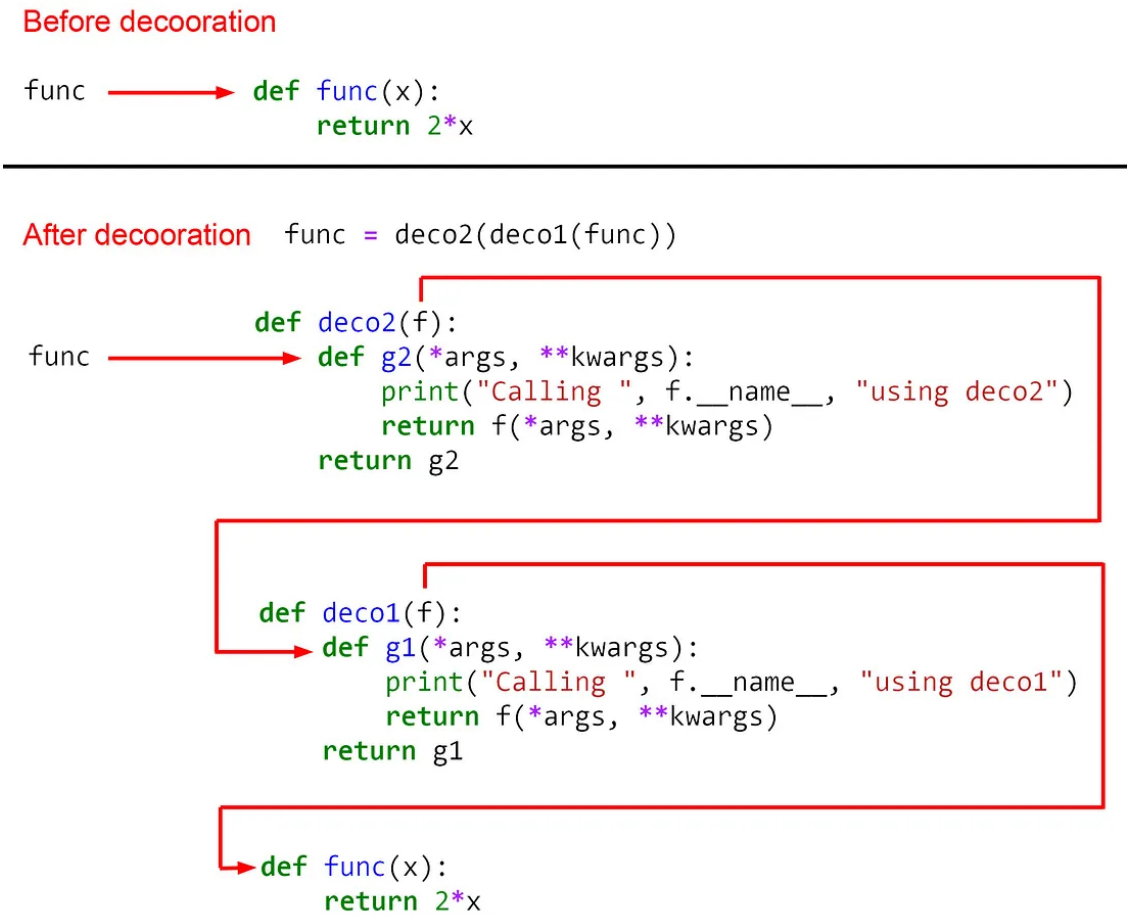
```
Calling  g1 using deco2
Calling  func using deco1
```

[ ]: 4

Di sini kita pertama-tama menerapkan deco1 ke func dan kemudian menerapkan deco2 ke closure yang dikembalikan oleh deco1. Jadi kita memiliki dua dekorator bertumpuk. Pertama-tama kita menghias func dengan deco1 dan kemudian dengan deco2.

Nyatanya g2 dan g1 sekarang bertindak sebagai antarmuka untuk fungsi asli func(x) yang didekorasi. Untuk mencapai fungsi asli func(x), pertama-tama kita panggil func yang merujuk ke g2. Di dalam g2, parameter f mengacu pada g1. Jadi dengan memanggil f, kita memanggil g1, dan kemudian di dalam g1 kita bisa memanggil f untuk memanggil fungsi asli func(x).

[ ]:
```
display.Image(r"C:\Users\ardik\OneDrive\Pythonenv\Praktikum PBF\func4.jpg")
```

[ ]:


Kita juga dapat menggunakan sintaks pai untuk menerapkan dekorator bertumpuk ke suatu fungsi. Untuk melakukan itu kita dapat menulis:

[ ]:
```
def deco1(f):
    def g1(*args, **kwargs):
        print("Calling ", f.__name__, "using deco1")
        return f(*args, **kwargs)
```

```
    return g1
def deco2(f):
    def g2(*args, **kwargs):
        print("Calling ", f.__name__, "using deco2")
        return f(*args, **kwargs)
    return g2
@deco2
@deco1
def func(x):
    return 2*x
func(2)
```

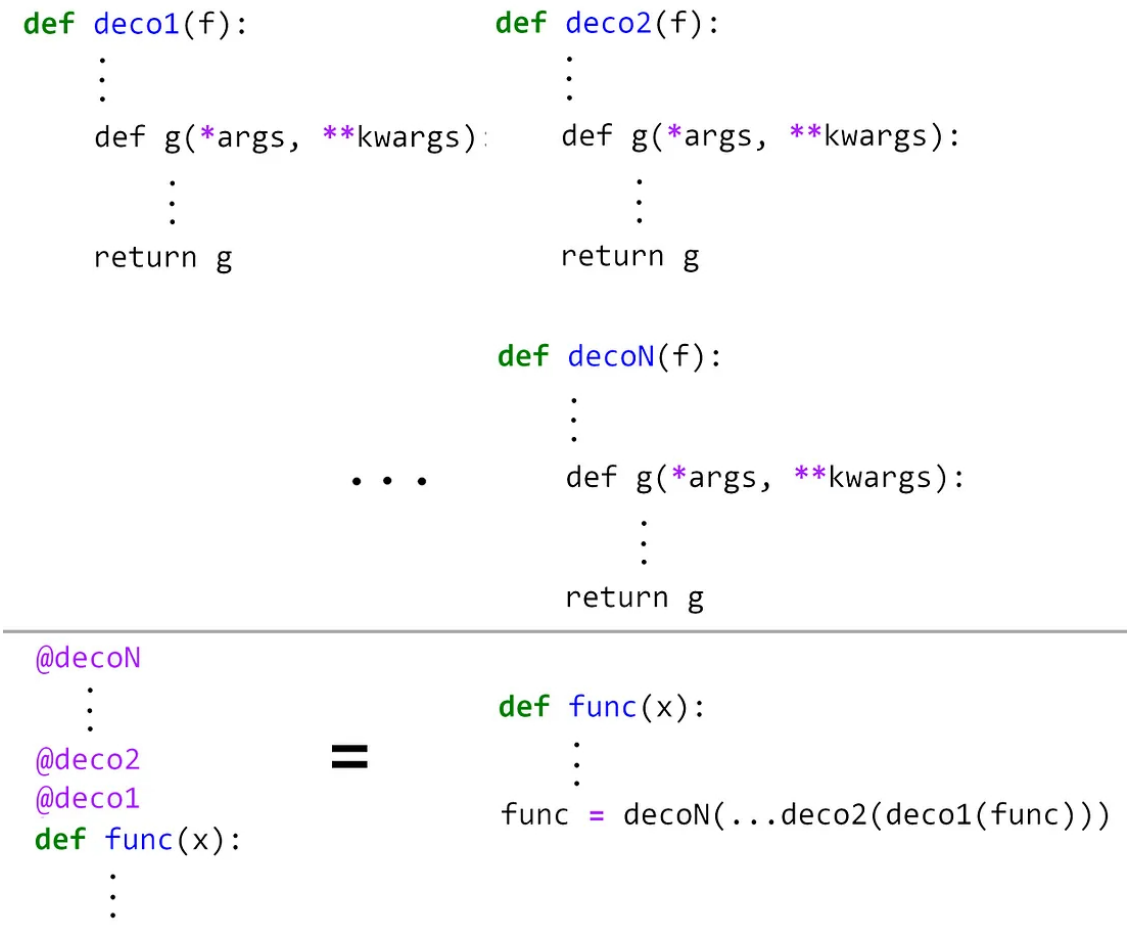
```
Calling  g1 using deco2
Calling  func using deco1
```

[ ]: 4

Di sini dekorator ditumpuk di atas definisi fungsi dalam urutan yang membungkus fungsi.

[ ]:
```
display.Image(r"C:\Users\ardik\OneDrive\Pythonenv\Praktikum PBF\func5.jpg")
```

[ ]:



Kita juga bisa menggunakan function compose yang diperkenalkan sebelumnya untuk menggabungkan kedua dekorator ini sebelum menerapkannya ke fungsi target. Operasi komposisi dapat mengambil dua fungsi deco1(f) dan deco2(f) dan menghasilkan fungsi deco sehingga deco(f) = deco2(deco1(f)). Seperti berikut:

[ ]:
```
def deco1(f):
    def g1(*args, **kwargs):
        print("Calling ", f.__name__, "using deco1")
        return f(*args, **kwargs)
    return g1
def deco2(f):
    def g2(*args, **kwargs):
        print("Calling ", f.__name__, "using deco2")
        return f(*args, **kwargs)
    return g2
deco = compose(deco2, deco1)
@deco
def func(x):
    return 2*x
func(2)
```

```
Calling  g1 using deco2
Calling  func using deco1
```

[ ]:
```
4
```

Sekarang kita bisa melihat contoh dekorator bertumpuk. Ingatlah bahwa kita mendefinisikan dekorator untuk memoisasi. Kita juga mendefinisikan dekorator untuk melacak fungsi rekursif. Sekarang kita dapat menggabungkannya untuk melacak fungsi rekursif dengan memoisasi. Pertama-tama kita menghias fungsi Fibonacci dengan dekorator memoize yang diberikan pada Contoh sebelumnya dan kemudian dengan dekorator jejak yang diberikan. Pertama-tama kita mencoba menghitung fib(5).

[ ]:
```
@trace
@memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
fib(5)
```

```
  memoized_func(5)
    memoized_func(4)
      memoized_func(3)
        memoized_func(2)
```

```
            memoized_func(1)
            1
            memoized_func(0)
            0
         1
         memoized_func(1)
         1
      2
      memoized_func(2)
      1
   3
   memoized_func(3)
   2
5
```

[ ]: 5

Karena kamus memo awalnya kosong, itu akan melakukan semua panggilan rekursif untuk menghitung hasilnya. Sekarang jika kita mencoba fib(6) kita mendapatkan:

[ ]:
```
fib(6)
```

```
memoized_func(6)
   memoized_func(5)
   5
   memoized_func(4)
   3
8
```

[ ]: 8

Kali ini hasil fib(5) dan fib(4) sudah tersimpan di kamus memo. Jadi hanya perlu mengambilnya untuk menghitung hasilnya, dan tidak perlu rekursi lagi. Anda mungkin memperhatikan bahwa nama fungsi fib tidak dicetak di pohon rekursi. Sebagai gantinya, kami melihat nama penutupan memoized_func. Alasannya adalah kita memiliki dua dekorator bertumpuk dan dekorator jejak menerima memoized_func sebagai parameternya, bukan fungsi fib. Jadi inside trace sekarang f merujuk ke `memoized_func`, dan `f.__name__` mengembalikan memoized_func. Di bagian selanjutnya, saya akan menunjukkan kepada Anda bagaimana kita dapat memecahkan masalah ini.