VERSION 2.1

27 OKTOBER 2024

[PEMROGRAMAN FUNGSIONAL]

Modul 4 – First Class Function

DISUSUN OLEH:

ALVIYA LAELA LESTARI

RAFLI KHARISMA AKBAR

DIAUDIT OLEH:

FERA PUTRI AYU L., S.KOM., M.T.

LAB. INFORMATIKA

UNIVERSITAS MUHAMMADIYAH MALANG

[PEMROGRAMAN FUNGSIONAL]

PERSIAPAN MATERI

1. Praktikan diharapkan telah memahami materi pada modul-modul sebelumnya.
2. Praktikan diharapkan juga mencari sumber belajar eksternal mengenai bahasa python.

TUJUAN PRAKTIKUM

Sub-CPMK 7: Mahasiswa mampu mendesain program dengan teknik yang tepat untuk menyelesaikan masalah dengan menggunakan paradigma pemrograman fungsional (P6)

TARGET MODUL

1. Mampu mengelola First Class Function secara fleksibel berdasarkan paradigma pemrograman fungsional
2. Mampu memahami dan menerapkan konsep lambda expression, closure, hingga decorator
3. Mampu mengelola dan memanfaatkan fitur-fitur paradigma pemrograman fungsional dalam menyelesaikan masalah komputasi

PERSIAPAN SOFTWARE/APLIKASI

1. Komputer/Laptop
2. Sistem operasi Windows/Linux/Mac OS/Android
3. Pycharm/Google Collab/ Jupyter Notebook
4. Source Code Google Colab Modul 4

1. First Class Function

Apa itu First Class Function? Seperti namanya yang dimana di kehidupan sehari-hari kita bisa mengetahui jika “First Class” mempunyai **privilege khusus** seperti bebas akses terhadap resource atau diberikan **hak untuk menjadi apapun** yang dia inginkan. Dalam lingkup fungsi, “First Class” berarti fungsi dapat diperlakukan sebagai warga negara kelas satu yang memiliki hak untuk menjadi apapun dalam bahasa pemrograman.

Ini berarti bahwa fungsi **dapat ditetapkan ke variabel**, diteruskan **sebagai parameter/argumen** di fungsi lain, dikembalikan dari fungsi **sebagai return value**, dan disimpan dalam struktur data, **sama seperti tipe data lain**nya (misalnya, angka, string, objek). Selain itu, dalam konsep First Class Function terdapat sebuah privilege untuk membuat fungsi tanpa nama (anonymous) dengan menggunakan **lambda.** Mari kita simak contoh-contoh berikut agar kita bisa memahami tentang First Class Function.

1.1 Fungsi dapat ditetapkan sebagai nilai ke variabel

Disini kita memperlakukan fungsi layaknya sebuah nilai atau data. Sehingga fungsi dapat digunakan dengan cara yang sama seperti nilai lainnya. Kita dapat membuat fungsi dan menetapkannya ke variabel seperti ini:

```python
# Fungsi untuk menghitung akar kuadrat. Dimana value merupakan bilangan
real positif.

def square_root(value):
    if value < 0:
        return ValueError("value tidak bisa kurang dari nol coyy!")
    return value ** 0.5

func_root = square_root
print(type(func_root))
print(func_root(16))
```

```
<class 'function'>
4.0
```

Dalam contoh di atas, kita bisa melihat bahwa fungsi square_root dideklarasikan dan disimpan di dalam variabel func_root sebagai referensinya. Ini disebut variabel fungsi. Variabel fungsi menyimpan referensi terhadap fungsi asli, yang berarti variabel tersebut menunjuk pada fungsi, namun fungsi tersebut belum dieksekusi.

Perhatikan bahwa dalam deklarasi seperti func_root = square_root, tidak ada tanda kurung () setelah nama fungsi. Ini berarti kita tidak mengeksekusi fungsi tersebut, melainkan hanya menyimpan referensi fungsi square_root di dalam variabel func_root. Jika kita coba cetak variabel tersebut, yang muncul adalah informasi bahwa func_root adalah sebuah objek fungsi, namun fungsi tidak dieksekusi.

Jika kita ingin mengeksekusi fungsi melalui variabelnya, kita harus menambahkan tanda kurung (), seperti func_root(). Tanda kurung tersebut memberi instruksi agar fungsi dijalankan (kita kenal sebagai istilah *function call*/pemanggilan fungsi), baru kemudian hasil eksekusi fungsi tersebut akan dikembalikan.

1.2 Fungsi sebagai parameter

Sebagaimana yang sudah kita implementasi di modul 3 pada topik map, filter, dan reduce. Kita membuat sebuah fungsi untuk dikirim sebagai parameter dari fungsi map, filter, dan reduce. Membuktikan bahwa fungsi dapat diteruskan sebagai parameter ke fungsi lain. Hal ini memungkinkan kode yang lebih fleksibel dan dapat digunakan kembali.

Disini kita akan mencoba membuat sendiri fungsi jenis ini. Alih-alih menggunakan fungsi built-in seperti saat di modul 3. Seperti ini contohnya:

```
''' fungsi untuk melakukan operasi + terhadap dua argumen x dan y
pre-cond: x dan y merupakan sembarang data, dengan syarat harus
setipe'''
def tambah(x,y):
    return x + y

'''fungsi apply_function(func, x, y) menerima sebuah fungsi func
untuk dieksekusi menggunakan x dan y sebagai nilai kembalian
pre-cond: func merupakan fungsi dua argumen;
          x,y disesuaikan dengan kriteria func pada argumen pertama'''
def apply_function(func, x,y):
    return func(x,y)

# Pastikan fungsi yang ingin dijadikan parameter sudah kita buat
terlebih dahulu
hasil = apply_function(tambah, 10, 20)
print(hasil)
```

30

1.3 Fungsi sebagai return value

Untuk mengimplementasikan hal ini, pertama-tama kita perlu mendefinisikan sebuah fungsi pertama yang akan berisi logika untuk mengembalikan sebuah nilai. Adapun nilai yang di return oleh fungsi seringkali berasal atau berada di dalam lingkup fungsi tersebut. Sehingga jika kita ingin mengembalikan sebuah fungsi sebagai return value, maka **kita harus mendefinisikan fungsi** kedua yang akan di return **di dalam fungsi** pertama tadi. Dengan kata lain, kita harus membuat fungsi bersarang (nested) disini. Perhatikan contoh berikut:

```python
def outer_function():
    def inner_function():
        return "Hello from inner function!"
    return inner_function

inner = outer_function()
print(inner())
```

```
Hello from inner function!
```

Disini kita kembali memperlakukan fungsi sebagai nilai/value seperti pada topik bahasan 1.1. dimana yang kita return adalah variabel fungsi (tanpa tanda()) dan bukan function call. Mengembalikan fungsi dari fungsi lain adalah teknik yang kuat yang memungkinkan perilaku dinamis seperti *enkapsulasi/abstraksi* dan kustomisasi. Pendekatan ini adalah fitur kunci dalam pemrograman fungsional, memungkinkan konsep seperti *callback* dan *closure*.

1.4 Fungsi di dalam Struktur Data

Sama hal nya dengan fungsi sebagai variabel. Fungsi juga dapat disimpan dalam struktur data seperti list, tuple, dan dict, atau struktur data lainnya, memungkinkan perilaku yang lebih kompleks.

```python
functions = [tambah, lambda x, y: x - y, outer_function(), func_root]

print(functions[0](10,5))
print(functions[1](1000,999))
print(functions[2]())
print(functions[3](16))
```

```
15
1
Hello from inner function!
4.0
```

Mari kita bedah code diatas (untuk *lambda* akan dibahas di poin selanjutnya). Dalam list functions terdapat beberapa fungsi yang sudah kita buat pada poin sebelumnya, dimana setiap index mewakili fungsi yang berbeda. Jika kita teliti lebih dalam terdapat perbedaan antara pemanggilan fungsi tambah, func_root dan outer_function(). Yaitu hanya outer_function yang memiliki tanda kurung sedangkan yang lain tidak. Kenapa bisa berbeda? Dalam contoh 1.1 sudah dijelaskan bahwa saat kita mendeklarasikan fungsi square_root ke dalam sebuah variabel func_root maka variabel tersebut akan menjadi referensi/nama pengganti fungsi yang dideklarasikan. Sehingga jika kita mencoba untuk mengetahui output seperti print(functions[3]) tanpa tanda kurung seperti di bawah ini :

```
print(functions[3])
```

maka output yang muncul akan menjadi :

```
<function square_root at 0x78b16cb55630>
```

Ini menandakan bahwa fungsi yang ada di dalam funct_root tidak tereksekusi dan hanya mengembalikan definisi/tipe dari objek tersebut. Maka dari itu kita perlu menambahkan tanda kurung sebagai perintah bahwa fungsi tersebut tidak hanya dipanggil namun juga perlu untuk kita eksekusi. Tanda kurung juga tetap sebagai media untuk mengisi argument seperti biasanya.

Namun, mengapa saat memanggil outer function kita memerlukan 2 tanda kurung di dalam list yaitu outer_function() dan saat memanggil yaitu functions[2](). Ini terjadi karena fungsi tersebut merupakan fungsi bersarang/nested. Maka dari itu, kita juga perlu dua kali pemanggilan fungsi dengan menggunakan 2 tanda kurung untuk mendapatkan hasil eksekusinya. Berikut beberapa contohnya :

```
# list functions menyimpan berbagai data fungsi sesuai dengan indexnya
functions = [tambah, lambda x, y: x - y, outer_function(), func_root]

# mengakses berdasarkan index
print(functions[2]())
print(outer_function()())

inner = outer_function() # merujuk pada contoh 1.2
print(inner())
```

2. Lambda Expression

Masih membahas tentang First Class Function, lambda merupakan salah satu privilege fungsi**.** Disini kita menggunakan kata expresi daripada fungsi. Karena istilah ekspresi lebih fungsional daripada istilah fungsi secara umum. Kenapa demikian? Karena sebuah expresi secara general membutuhkan sebuah input (trigger) dan pasti akan menghasilkan sesuatu sebagai output atau pesan dari expresi itu sendiri.

Lambda expression adalah cara untuk membuat fungsi kecil tanpa nama *(anonymous function).* Lambda expression cocok digunakan untuk **membuat fungsi yang hanya digunakan sekali atau dalam waktu singkat.** Dalam penggunaannya, kita cukup menggunakan kata kunci **lambda** diikuti oleh nama variabel atau parameter yang digunakan sebagai input (daripada menggunakan kata kunci 'def' untuk mendefinisikan suatu fungsi) dan expression sebagai output atau return value dari lambda.

Berikut ini adalah struktur kode penggunaan lambda expression:

```
lambda parameter: expression
```

Contoh penggunaan:

```
# Daripada kita membuat fungsi sekali pakai seperti ini:
def pengurangan(a, b):
  return a - b

hasil_def = pengurangan (10, 4)
print("Hasil: ", hasil_def)

# Lebih efisien kita menggunakan lambda expression:
hasil_pengurangan = lambda a, b: a - b
print("Hasil: ", hasil_pengurangan(10, 4))
```

Dengan menggunakan lambda expression, kode terlihat lebih ringkas dan caranya juga lebih mudah. Kalian cukup mengambil parameter dan expresi (yang seringkali

digunakan sebagai return value) dalam fungsi dan menuliskannya kembali sebagai parameter dan expresi dari lambda.

Apa arti dari variabel a dan b pada kode diatas? a dan b adalah parameter / argumen yang akan digunakan saat melakukan operasi menggunakan lambda. Untuk parameter/argument disini, kita bisa menamakan sesuka hati kita. Namun, biasanya penggunaan nama variabel yang sederhana seperti x dan y lebih readable karena tujuan utama dari lambda adalah fungsi sederhana.

Karena lambda adalah fungsi tanpa nama, maka kita perlu menyimpannya dalam variabel. Agar memudahkan kita untuk mengaksesnya. Atau bisa juga secara langsung dikirim sebagai parameter fungsi (untuk digunakan bersamaan dengan fungsi map/filter).

```
# menggunakan lambda secara langsung sebagai parameter fungsi 'filter dan map'
data = filter(lambda x: x%2, range(10)) #filter data ganjil range(10)
hasil = map(lambda x: x*2, data)  #mapping x*2 untuk semua data

print(list(hasil))
```

```
[2, 6, 10, 14, 18]
```

Berikut contoh lain dari inisialisasi argumen dalam lambda :

a.
```
# lambda expression untuk mengalikan bilangan dengan 2

kali_2 = lambda x: x * 2
print(kali_2(5))
```

```
10
```

b.
```
# lamda expression yang mengambil 2 argumen untuk dijumlahkan

add = lambda x, y: x + y

var1 = 3
var2 = 4
result = add(var1, var2)
```

```
7
```

c.
```
# lamda expression yang mengambil 2 pareameter
# angka (3, 4) akan menjadi argumen yang mengisi lambda

result = (lambda x, y: x + y)(3, 4)
print(result)
```

```
7
```

Dalam contoh “c” terdapat implementasi konsep lain dari pemrograman fungsional yaitu **currying**, Currying adalah teknik dalam pemrograman fungsional di mana sebuah fungsi yang mengambil beberapa argumen diubah menjadi urutan fungsi yang masing-masing mengambil satu argumen. Proses ini memungkinkan kita untuk memecah fungsi yang menerima beberapa argumen menjadi beberapa fungsi yang lebih sederhana yang dapat dieksekusi secara bertahap. Currying juga bisa dianggap sebagai metode/cara lain untuk mengeksekusi nested function secara langsung. Perhatikan contoh kode pada bab 1.3 sebelumnya, yang mengandung fungsi bersarang:

```
1 def outer_function():
2     def inner_function():
3         return "Hello from inner function!"
4     return inner_function
```

fungsi tersebut bisa juga kita panggil secara langsung dengan metode currying seperti berikut:

```
outer_function()()
```

```
'Hello from inner function!'
```

Perlu kita ketahui bahwa lambda sama halnya dengan fungsi namun **tidak semua** sifat fungsi bisa dilakukan oleh lambda. Berikut batasan yang tidak bisa dilakukan oleh lambda :

1. Tidak bisa leluasa menggunakan if, for dan while (khusus if hanya keadaan ekspresi kondisional sederhana yang bisa)
2. Lambda adalah fungsi anonim sehingga kita tidak bisa menamai langsung, yang bisa kita ubah adalah variabel yang mendeklarasikan lambda
3. Lambda tidak relevan dengan kode dengan kompleksitas tinggi. Namun, lambda sangat cocok untuk fungsi inline.

3. Higher Order Function

Pada bahasa pemrograman fungsional yang mendukung First Class Function (seperti Python) terdapat istilah High Order Function (HOF). HOF merupakan fungsi yang **menerima sebuah fungsi sebagai parameter input** dan atau fungsi yang **mengembalikan sebuah fungsi sebagai return value.** Dua hal tersebut merupakan

bagian dari First Class Function. Sehingga, dengan kata lain, HOF merupakan hasil implementasi dari First Class Function.

3.1 Menerima sebuah fungsi sebagai parameter input

```python
# Contoh fungsi sebagai argumen/input dari fungsi lain

def kali (a,b):
  return a * b

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

def kurang(a, b):
  return a - b

def hitung(a, operasi, b):
  return operasi(a, b)

tiga_tambah_dua = hitung(3, tambah, 2)
print("Hasil: ", tiga_tambah_dua)

dua_kali_lima = hitung(5, kali, 2)
print("Hasil: ", dua_kali_lima)

sebelas_kurang_dua = hitung(11, kurang, 2)
print("Hasil: ", sebelas_kurang_dua)
```

Dalam kode di atas, fungsi tambah dijadikan sebagai parameter input dari fungsi 'hitung', yang kemudian ditampung pada variabel 'tiga_tambah_dua'. Demikian halnya dengan 'dua_kali_lima' dan 'sebelas_kurang_dua'.

Dalam bahasa Python, terdapat built-in Higher Order Function (HOF) atau fungsi-fungsi yang sudah disediakan oleh python untuk memudahkan kita. Sebenarnya kita sudah mengenal contoh-contoh dari Built-in HoF pada modul sebelumnya seperti **map(), filter(),** dan **reduce()**. Nah, di modul ini kita akan mengimplementasikan lambda dalam HoF agar kode program kita menjadi lebih ringkas dan sederhana. Sebagaimana konsep dari paradigma fungsional itu sendiri. Penggunaan Built-In Function memudahkan kita dalam membuat program karena mereka sejenis dengan inline function sehingga membuat kinerja kode kita lebih bersih dan efisien.

3.1.1 Map

```
#1. Menyiapkan fungsi transformasi
'''
desc: fungsi double(x) untuk MERUBAH input x menjadi 2x
pre-cond: input x merupakan sembarang data, bisa int, float, dlsb
post-cond: 2x sebagai return value
'''
def double(x):
    return x * 2

#2. Data berupa sequence/iterable data
numbers = [1, 2, 3, 4, 5]

#3. Penggunaan fungsi map
doubled_numbers = map(double, numbers)
```

Jika pada modul 3 sebelumnya kita perlu mendeklarasikan sebuah fungsi transformasi untuk digunakan dalam map seperti pada contoh kode diatas. Sekarang mari kita lihat implementasi dari penggunaan map dengan lambda berikut:

```
# Menggunakan fungsi map untuk mengalikan
setiap elemen dengan 2
numbers = [1, 2, 3, 4, 5]
result = map(lambda x: x * 2, numbers)

print(list(result))
```

Coba kalian bandingkan dengan kode sebelumnya! Dalam contoh diatas kita mengimplementasikan konsep dari First Class dan juga High Order Function. Disini anonymous function lambda menjadi argumen dalam built-in HOF map.

Pada kode di atas, fungsi map() mengambil dua argumen, yaitu sebuah fungsi (lambda x: x * 2) dan sebuah iterable (numbers). Setiap elemen dalam list numbers dikirim sebagai argumen x ke dalam lambda. Lambda tersebut akan mengalikan nilai x dengan 2 dan me-return hasilnya. Dengan kata lain, map() menerapkan operasi lambda ini ke setiap elemen dalam list numbers, menghasilkan daftar baru dengan setiap elemennya dikalikan 2. Hasil dari map() adalah sebuah map object, yang kemudian dikonversi menjadi list agar dapat dicetak.

Kita juga bisa memisah lambda ke dalam variabel lain untuk reusability code. Seperti berikut:

```
# Menggunakan fungsi map untuk mengalikan
setiap elemen dengan 2
numbers = [1, 2, 3, 4, 5]
kali = lambda x: x * 2
result = map(kali, numbers)

print(list(result))
```

3.1.2 Filter

`filter` adalah salah satu higher-order function bawaan di Python yang digunakan untuk **menyaring** elemen-elemen dari suatu iterable data (seperti list, tuple, atau set) **berdasarkan kondisi tertentu**. Fungsi **`filter`** menerima dua argumen utama:

1. **function**: Fungsi yang menentukan kondisi atau kriteria untuk menyaring elemen-elemen dari iterable. Fungsi ini harus mengembalikan nilai boolean (`True` atau `False`). Hanya elemen-elemen yang membuat fungsi ini mengembalikan `True` yang akan dimasukkan ke dalam hasil akhir.
2. **iterable**: Sekumpulan data (misalnya, list, tuple, set) yang akan disaring.

Hasil dari `filter` adalah sebuah objek filter (sejenis iterator) yang dapat dikonversi menjadi list, tuple, atau struktur data lain. Perhatikan contoh kode pemanfaatan fungsi filter berikut:

```
#1. Kita siapkan contoh code dari modul 3

def fungsi_genap(x):
    return x % 2 == 0

#2. Data yang akan di filter
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

#3. Pengunaan fungsi filter()
angka_genap = filter(fungsi_genap, numbers)
print("Hasil filter biasa : ", list(angka_genap))
```

```
#4. Menggunakan Lambda
angka_genap_lambda = filter(lambda x: x % 2 == 0, numbers)
print("Hasil Lambda : ", list(angka_genap_lambda))
```

`filter` memisahkan logika penyaringan dari implementasi loop, membuat kode lebih bersih dan lebih mudah dipahami.

```
data = ["kelapa", "apel", "kiwi", "jeruk",
"kacang"]

# mem-filter elemen yang dimulai dengan huruf "a"
hasil = filter(lambda x: x.startswith("a"), data)

print(list(hasil))  # Output: ['apel']
```

Dalam contoh diatas kita bisa melihat bahwa kita bisa membuat inline function atau one line function. Variabel 'hasil' akan berupa objek filter yang lazy. Berisi data hasil penyaringan yang berawalan huruf 'a'. Perlu diingat kembali bahwa hasil dari lazy evaluation tidak dapat langsung di-print. Untuk pengecekan hasil, kita dapat melakukan parsing data ke dalam list, tuple, atau set sebelum memanggil fungsi print().

3.1.3 Reduce

`reduce` adalah fungsi higher-order yang digunakan untuk mengakumulasi hasil dari iterasi elemen-elemen dalam iterable (seperti list atau tuple) menjadi satu nilai tunggal. Fungsi ini berada dalam modul `functools` di Python. Berikut adalah contoh implementasi `reduce` tanpa dan dengan `lambda`:

```
from functools import reduce
'''
desc: fungsi untuk menghitung hasil perkalian dari dua argumen a dan b
pre-cond: salah satu argumen a/b harus berupa number
post-cond: hasil dari a*b sebagai nilai kembalian
'''
def perkalian(a,b):
  return a*b

print(reduce(perkalian,range(1,6)))

fungsi_lambda = reduce(lambda x,y: x * y, range(1,6))

print("Ini adalah hasil lambda : ", fungsi_lambda)
```

Perbedaan reduce dengan map/filter terdapat pada output/return value yang dihasilkan. Ketika map/filter mengembalikan sebuah iterable baru, output dari reduce adalah sebuah nilai tunggal. Selain itu, input fungsi untuk reduce adalah **fungsi dua argumen.** Sehingga lambda yang dibuat dalam reduce juga harus menerima 2 argumen (x dan y), dimana map/filter hanya membutuhkan fungsi 1 argumen (lambda x).

Tabel Perbedaan antara Map, Filter, dan Reduce

Aspek	map()	filter()	reduce()
Tujuan	Menerapkan fungsi transformasi ke setiap elemen dalam iterable	Menerapkan fungsi logika untuk menyaring elemen-elemen dalam iterable	Mengurangi iterable menjadi satu nilai tunggal dengan fungsi kumulatif
Modul	Built-in	Built-in	functools
Argumen Fungsi	Fungsi dengan satu argumen (elemen dari iterable)	Fungsi dengan satu argumen (elemen dari iterable)	Fungsi dengan dua argumen (akumulator & elemen berikutnya)
Hasil Akhir	Map object (lazy evaluation)	Filter object (lazy evaluation)	Nilai tunggal
Iterasi	True	True	False (hanya mengembalikan satu nilai)
Panjang Hasil	Sama dengan panjang iterable asli	Bisa lebih pendek dari iterable asli	Satu nilai tunggal
Contoh Penggu-naan	Menerapkan fungsi untuk menggandakan setiap elemen dalam list/iterable lain	Menyaring angka-angka genap dari list/iterable lain	Menjumlahkan semua angka dalam list/iterable lain

3.2 Mengembalikan sebuah fungsi sebagai return value

```
def operasi(n):
  def tambah(x):
    return n + x
  return tambah

tambah_5 = operasi(5)
print("Hasil: ",
tambah_5(7))
```

Dalam kode di atas, fungsi 'operasi' menerima satu argumen 'n' dan mengembalikan sebuah fungsi bernama 'tambah' sebagai nilai kembalian (bukan function call). **Agar dapat dikirim sebagai return value, maka fungsi tambah perlu didefinisikan di dalam fungsi sebagai bagian dari fungsi (inner function)**. Fungsi 'tambah' ini, pada gilirannya, menerima satu argumen 'x' dan mengembalikan hasil penjumlahan dari 'n' dan 'x'.

4. Closure

Apa itu closure? Closure disini bukan salah satu lagu karya Pamungkas ya. Dalam pemrograman fungsional closure adalah istilah umum untuk deklarasi fungsi di dalam fungsi (*nested function)*. Masih bingung, simak contoh berikut:

```
def outer_function(message):
    '''
    outer_function menerima sebuah pesan dan menyimpannya dalam variabel lokal `localvar`.
    Fungsi ini mengembalikan inner_function, yang merupakan closure karena menangkap `localvar`.
    '''
    localvar = message
    def inner_function():
        '''inner_function mencetak nilai `localvar` yang disimpan di outer_function'''
        print(localvar)
    return inner_function

# Membuat closure dengan memanggil outer_function dan menyimpan fungsi yang dihasilkan
closure = outer_function("Hello, I'm under the water!")
closure()  # Memanggil closure, yang akan mencetak pesan yang disimpan
```

```
Hello, I'm under the water!
```

Apakah berarti sama halnya dengan konsep First Class Function? Jelas berbeda, karena First Class Function adalah sifat dari sebuah fungsi seakan fungsi adalah “First Class Citizens” sedangkan Closure, HoF dsb, merupakan implementasi dari sebuah konsep First Class Function.

Tentang closure, pada pemanggilan fungsi outer_function(), isi dari variabel message, yaitu 'Hello, I'm under the water' masih tetap diingat meskipun kita sudah selesai dengan pemanggilan fungsi outer_function()/sudah return. Padahal pada fungsi biasa, seharusnya variabel fungsi akan terhapus begitu return/eksekusi terhadap fungsi selesai. Masih belum paham? coba jalankan kode berikut

```
1 del outer_function
2
3 closure()
```

```
Hello, I'm under the water!
```

Ketika kita menghapus fungsi outer_function dan kita memaksa untuk memanggil closure yang mana sudah kita deklarasikan sebelumnya, maka program akan tetap berjalan seolah variabel closure masih memiliki memori dan mengingat bahwa dia dideklarasikan sebagai "pengganti" dari fungsi outer_function jadi ketika main functionnya dihapus dia masih tetap bisa berjalan.

Closure dinamakan demikian karena fungsi "menutup" lingkup di mana ia didefinisikan dan "mengikat" variabel-variabel dari lingkup tersebut, sehingga dapat mengakses dan menggunakan variabel-variabel tersebut meskipun dieksekusi di luar lingkup aslinya. Ini memungkinkan fleksibilitas dan modularitas yang lebih besar dalam pemrograman.

Masih pusing? simak **analogi sederhana** berikut untuk **teori closure**:

Bayangkan Anda memiliki sebuah ruangan (lingkup luar) dengan beberapa barang di dalamnya (variabel). Anda membuat sebuah kotak (fungsi dalam) dan memasukkan beberapa barang dari ruangan ke dalam kotak tersebut. Ketika Anda membawa kotak keluar dari ruangan, kotak tersebut masih berisi barang-barang dari ruangan tersebut. Anda bisa membuka kotak kapan saja untuk melihat atau menggunakan barang-barang di dalamnya, meskipun Anda sudah keluar dari ruangan. Untuk contoh lebih dalam program bisa diakses di collab ya!

5. Decorator

Decorator adalah fungsi yang mengambil fungsi lain sebagai argumen dan mengembalikan fungsi baru dengan menambahkan atau memodifikasi perilaku dari fungsi asli tanpa mengubah kodenya. Dari penjelasan ini mungkin terdengar tidak asing,

yap ini merupakan implementasi dari sifat First Class yaitu fungsi sebagai argumen atau sebagai return value. Dengan kata lain sama hal nya dengan higher order function/HOF. Perhatikan contoh berikut:

```
# ini akan menjadi dekorator
def halo_decorator(func):
  def halo():
    print('gimme my moneyy')
    func()
    print("yayyyyyy")
  return halo

def myname():
  print("ditegor kiwil")

contoh_panggil = halo_decorator(myname)
contoh_panggil()
```

```
gimme my moneyy
ditegor kiwil
yayyyyyy
```

Secara sekilas, contoh diatas tidak ada beda dengan HOF. Dimana Fungsi halo_decorator() menerima input sebuah fungsi dan juga mengembalikan fungsi inner sebagai return value. Selain itu, fungsi halo_decorator() membungkus fungsi myname dan menambahkan perintah print sebelum dan sesudah fungsi tersebut dipanggil. Dekorator memungkinkan kita menambahkan atau memodifikasi perilaku fungsi tanpa mengubah fungsi aslinya, memberikan fleksibilitas dalam mengelola alur eksekusi program. Saat contoh_panggil() dijalankan, dekorator pertama mencetak "gimme my moneyy", lalu menjalankan myname() yang mencetak "ditegor kiwil", dan terakhir mencetak "yayyyyyy".

Namun, contoh diatas belum sepenuhnya menggunakan dekorator. Dalam python terdapat simbol “@” - ini bukan simbol yang menunjukkan override atau note seperti di Java ya. Dalam python simbol tersebut dinamakan decorator. Kita cukup menambahkan simbol @ dan diikuti dengan nama fungsi dekorator di atas deklarasi fungsi yang akan kita dekorasi. Seperti ini contohnya:

```
@halo_decorator
def myname():
  print("ditegor kiwil")

myname()
```

```
gimme my moneyy
ditegor kiwil
yayyyyyy
```

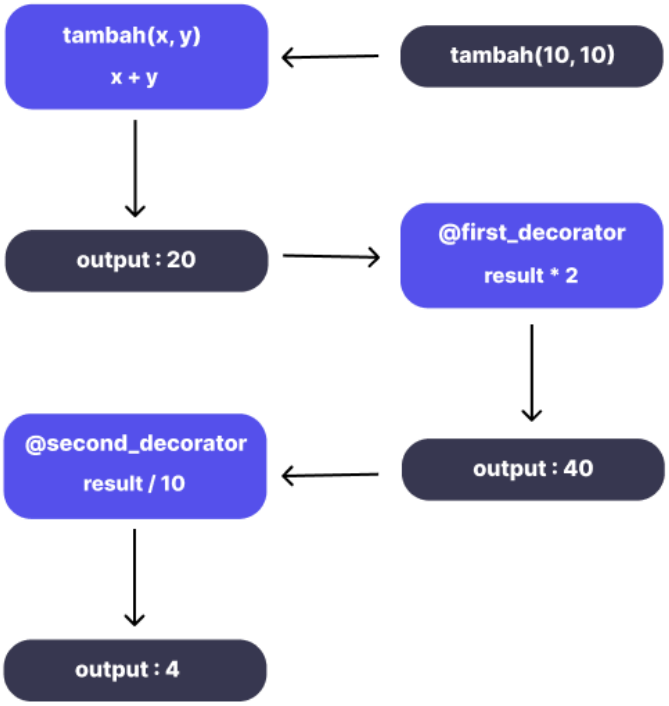
Dalam kondisi diatas konsep yang diimplementasikan sama dengan konsep sebelumnya, dimana fungsi yang memiliki "@" diatasnya akan menjadi argumen sesuai dengan fungsi mana dia akan dilempar.

Jadi, kedua hal ini memiliki tujuan dan konsep decorator yang sama hanya saja secara penulisan lebih efisien ketika kita menggunakan "@" karena kita tidak perlu mendeklarasikan ulang fungsi ke dalam variabel baru, fungsi yang memiliki decorator akan "otomatis" terlempar (menjadi argumen) di fungsi decorator aslinya (halo_dekorator()). Lalu apakah mungkin sebuah fungsi memiliki lebih dari 1 decorator? simak contoh lain berikut:

```python
# Fungsi ini akan menjadi decorator yang pertama
def first_decorator(func):
    def wrapper(x, y):
        result = func(x, y)
        return result * 2
    return wrapper

# Fungsi ini akan menjadi decorator yang kedua
def second_decorator(func):
    def wrapper(x, y):
        result = func(x, y)
        return result / 10
    return wrapper

@second_decorator
@first_decorator
def tambah(x, y):
    return x + y

print(tambah(10, 10))
```

```
4
```

Kenapa Outputnya hanya ada 1 yaitu 4? Ingat decorator diatas bukan halnya seperti kita memanggil fungsi biasa namun, kita menjalankan second_decorator() dengan argumen yang sudah diproses/dieksekusi (return value) pada fungsi decorator pertama yaitu first_decorator(), first decorator mengambil argumen dari hasil return value fungsi tambah(). Masih bingung mari simak diagram berikut sebagai alur kerja program nested dekorator sebelumnya:



Jika masih ada yang belum dipahami, silahkan bertanya kepada asisten lab yaa! Atau langsung ke dosen juga boleh.. Feel free to ask.

CODELAB

CODELAB 1

```python
from functools import reduce

nilai_mahasiswa = {
    'Zaidun' : 99,
    'Suwarsono' : 100,
    'Dedi' : 75,
    'Diki' : 88,
    'Joko' : 40,
    'Rusdi' : 78,
    'Kusnadi' : 93,
    'Diki' : 100,
    'Ansori' : 92,
    'Andri' : 76,
    'Salsa' : 58
}

total_nilai = ...
nilai_tertinggi = ...
mahasiswa_lulus = ... # mahasiswa lulus kkm = 75
# tambah 5 poin untuk mahasiswa yang di bawah 75
nilai_mahasiswa_update = dict(map()) # lanjutkan hintnya

# tampilkan nilai
```

Isi variabel kosong diatas dengan one line function menggunakan lambda, map, filter ataupun reduce. Berkreasi Lah!

CODELAB 2 [LIVE CODE]

Ubahlah fungsi-fungsi yang kalian buat pada **codelab modul 3** menjadi bentuk lambda!

CODELAB 3

1. Buatlah decorator bernama ‘simple decorator’ yang di dalamnya terdapat wrapper (inner function)
2. Decorator tersebut akan mencetak “Fungsi ini dipanggil” sebelum mencetak “Hello World”
3. Gunakan decorator tersebut pada sebuah fungsi bernama `hello` yang hanya mencetak "Hello, World!".

Contoh output :

```
Fungsi ini dipanggil

Hello, World!
```

TUGAS PRAKTIKUM

TUGAS 1

Perhatian! Kita akan menggunakan referensi tugas dari modul sebelumnya.

1. Dari tugas CRUD yang telah kalian kembangkan dari modul 1-3, kembangkanlah fungsi-fungsi dan program yang kalian buat sebelumnya dengan konsep **First Class Function (Lambda Expression, Higher Order Function disertai penggunaan Built in HoF).** Pastikan program yang kalian kembangkan memenuhi konsep yang telah diperintahkan.
2. Dalam program CRUD tersebut, buatlah sebuah fungsi baru (fitur bebas) atau mengubah fungsi yang ada dengan menambahkan **decorator**
3. Implementasikan penggunaan **Closure** dalam CRUD yang anda kembangkan.

   PS: Kalian tidak perlu mempertahankan program menu yang prosedural lagi disini (main func). Fungsi-fungsi dalam paradigma fungsional dapat langsung dijalankan (testing) secara langsung dan terpisah.

TUGAS 2 (INTERMEDIATE)

1. Buatlah sebuah inputan untuk menyimpan nilai mahasiswa ke dalam sebuah list (inputan mendukung tipe data float)

```
contoh_input = 7.5,8.2,9,6.8,5
contoh_output = [7.5, 8.2, 9.0, 6.8, 5.0]
```

2. Dari nilai mahasiswa tersebut buatlah sebuah fungsi yang mengimplementasikan **lambda** dan built in **HoF** untuk menghitung:
    a. rata-rata,
    b. jumlah nilai yang di bawah rata-rata (jika **nim ganjil**) dan nilai yang di atas rata-rata (untuk **nim genap)**

3. Buatlah sebuah decorator untuk menghitung waktu eksekusi (menggunakan library time) dari sebuah fungsi. Dan implementasikan dekorator tersebut pada setiap fungsi yang kalian buat sebelumnya (dalam tugas ini).

)Note: Implementasikan materi modul 4 ke dalam program anda.

TUGAS 2 (ADVANCED)

1. Buatlah sebuah inputan user yang menerima string angka yang dipisahkan koma, selanjutnya ubah dan petakkan inputan menjadi titik-titik (x,y) seperti berikut:

```
contoh_input = 1,2,3,4,5,6,7,8,9,10,11
contoh_output = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

)Note: lakukan exception ketika inputan berjumlah ganjil.

2. Buatlah 3 fungsi transformasi (translasi, rotasi, dilatasi) dengan memanfaatkan teknik HOF dan closure (nested function) terhadap daftar titik-titik (x,y) hasil dari soal no.1.
3. Dengan memanfaatkan fungsi yang sudah kalian siapkan pada soal no.2, lakukan transformasi terhadap semua titik hasil mapping dengan ketentuan sebagai berikut:
    I. pertama, translasi dengan tx = 3 dan ty = 7
    II. kemudian rotasi sejajar sumbu x positif sebesar 60 derajat
    III. terakhir, dilatasikan dengan faktor skala 1.5

IV. Tampilkan setiap hasil transformasi.

4. Buatlah sebuah decorator untuk menghitung waktu eksekusi (menggunakan library time) dari sebuah fungsi. Dan implementasikan dekorator tersebut pada setiap fungsi yang kalian buat sebelumnya (dalam tugas ini).

)Note: Implementasikan materi modul 4 ke dalam program anda.

KRITERIA & DETAIL PENILAIAN

KETERANGAN		POIN	PROGRAM IDENTIK
Codelab	Codelab 1, 2, & 3 masing-masing 5 poin	15	15
Tugas 1	Pemahaman dan Ketepatan program	20	21
	Code/Kelengkapan fitur	15	0
Tugas 2 Intermediate	Pemahaman dan Ketepatan program	20	21
	Code/Kelengkapan fitur	10	0
Tugas 2 Advance	Pemahaman dan Ketepatan program	30	31
	Code/Kelengkapan fitur	20	0
Total (Intermediate)		**80**	**57**
Total (Advance)		**100**	**67**

*)Note: Program Identik berarti program sama persis dengan praktikan lain sehingga yang dinilai hanya pemahaman terhadap materi (code tidak mendapat bobot nilai sama sekali).

**)Poin diatas merupakan poin maksimal yang bisa diperoleh. Asisten bisa memberikan nilai dibawah itu jika dirasa praktikan tidak maksimal saat demo (kurangnya pemahaman tentang apa yang di demokan).