VERSION 1.1

AGUSTUS 14, 2023

[PRAKTIKUM PEMROG. FUNGSIONAL]

MODUL 2 – Fitur-fitur karakteristik pemrograman fungsional (Effect-free programming)

DISUSUN OLEH :
Fildzah Lathifah
Hania Pratiwi Ningrum

DIAUDIT OLEH
Fera Putri Ayu L., S.Kom., M.T.

PRESENTED BY: TIM LAB-IT

UNIVERSITAS MUHAMMADIYAH MALANG

PERSIAPAN MATERI

- Praktikan diharapkan telah memahami konsep tipe data sequence pada bahasa python
- Praktikan diharapkan mempelajari dari sumber eksternal mengenai bahasa python, khususnya exception handling untuk menangani error yang mungkin terjadi dalam pengerjaan tugas praktikum.

TUJUAN PRAKTIKUM

- Praktikan dapat memahami dan mengimplementasikan konsep pemrograman fungsional basic (effect-free programming) menggunakan Bahasa pemrograman python
- Praktikan mampu memahami dan mengimplementasikan fitur fungsi murni, lambda, list comprehension dan generator
- Praktikan mampu menentukan teknik yang tepat dalam menyelesaikan masalah pemrograman fungsional perangkat lunak

---

TARGET MODUL

Penguasaan materi:

1. Pure Function
2. Lambda Expression
3. List Comprehension
4. Generator

PERSIAPAN SOFTWARE/APLIKASI

- Komputer/Laptop
- Sistem operasi Windows/Linux/Mac OS/Android
- Pycharm/Google Collab/ Jupyter Notebook

MATERI POKOK

Pada pemrograman fungsional, modul juga dapat diakses melalui google collab agar lebih interaktif : Modul 2

1. Pure Functions

Di modul 1 kita sudah belajar tentang tipe data tuple dan list, masih ingat kan kalau tuple itu sifatnya immutable atau kekal (tidak dapat diubah). Nah, dalam pemrograman fungsional, setiap fungsi yang dibuat sebaiknya bersifat immutable terhadap variabel lain di luar fungsi. Dengan kata lain, fungsi yang ada tidak boleh merubah dan juga tidak boleh terikat dengan variabel lain di luar fungsi. Fungsi tersebut dinamakan **pure function** atau fungsi murni. Hmm bingung ngga? Coba perhatikan contoh berikut:

Dalam contoh ini, fungsi yang pertama adalah 'calculate_total_stock' yang akan menghitung total stok barang dari daftar barang yang diberikan. Fungsi ini hanya bergantung pada parameter yang diberikan dan tidak mempengaruhi variabel di luar fungsi.

```
items = [
    {'name': 'Pensil', 'stock': 100},
    {'name': 'Buku', 'stock': 50},
]

def calculate_total_stock(items):
    total_stock = 0
    for item in items:
        total_stock += item['stock']
    return total_stock

# Menghitung total stok barang
total_stock = calculate_total_stock(items)
print(f"Total stok barang: {total_stock}")
```

```
Total stok barang: 150
```

Selanjutnya, fungsi kedua adalah 'add_stock' yang akan menambahkan stok barang tertentu ke dalam daftar barang. Fungsi ini akan memodifikasi variabel items yang ada di luar fungsi.

```
def add_stock(item_name, quantity):
    global items
    for item in items:
        if item['name'] == item_name:
            item['stock'] += quantity
            print(f"Stok {item_name} ditambahkan sebanyak {quantity}. Stok saat
ini: {item['stock']}")
            break
    else:
        print(f"Barang {item_name} tidak ditemukan dalam daftar.")

# Menambahkan stok barang
add_stock('Pensil', 50)

# Cek stok barang saat ini
total_stock = calculate_total_stock(items)
```

```
print(f"Total stok barang terbaru: {total_stock}"
```

```
Stok Pensil ditambahkan sebanyak 50. Stok saat ini: 150
Total stok barang terbaru: 200
```

Setelah mengamati kedua fungsi diatas, sudahkah kalian menemukan mana yang merupakan pure function?? Yaps betul, fungsi 'calculate_total_stock' adalah contoh dari pure function karena hanya bergantung pada parameter yang diberikan (items) dan tidak mempengaruhi variabel di luar fungsi. Sedangkan, fungsi 'add_stock' adalah contoh dari non-pure function karena memodifikasi variabel items yang berada di luar fungsi.

Apakah fungsi yang seperti itu (non-pure function) bisa diubah menjadi pure function? Coba amati skenario Mobile Legends Simple Game berikut ini. Dengan data dictionary hero yang ada, kita akan menyiapkan sebuah fungsi untuk memilih hero saat awal permainan:

```
allheroes = [{"Hero" : "Kagura", "Role" : "Mage", "DMG" : 350, "Gold" : 1000},
         {"Hero" : "Yve", "Role" : "Mage", "DMG" : 250, "Gold" : 1000},
        {"Hero" : "Lancelot", "Role" : "Assassin", "DMG" : 200, "Gold" : 1000},
        {"Hero" : "Hayabusa", "Role" : "Assassin", "DMG" : 300, "Gold" : 1000},
         {"Hero" : "Natalia", "Role" : "Assassin", "DMG" : 150, "Gold" : 1000},
         {"Hero" : "Cecilion", "Role" : "Mage", "DMG" : 200, "Gold" : 1000}]

def chooseHero():
 print("Select 1 Hero you want to play: ")
 selectedhero = {}
 for i in range(len(allheroes)):
   print("Id : {}".format(i))
   print("Detail : {}".format(allheroes[i]))
 inputs = input("silahkan masukkan pilihan Id: ")
 try:
   selectedhero.update(allheroes[int(inputs)])
   return selectedhero
 except:
   return "Wrong Input!"

def start():
 print("Welcome to mini Mobile Legends Game\n\n")
 hero = chooseHero()
 print("\nYour hero is ",hero)
 #skenario game berhenti disini lanjutannya belum dibuat

#mari jalankan gamenya
start()
```

```
Welcome to mini Mobile Legends Game
```

```
Select 1 Hero you want to play:
Id : 0
Detail : {'Hero': 'Kagura', 'Role': 'Mage', 'DMG': 350, 'Gold': 1000}
Id : 1
Detail : {'Hero': 'Yve', 'Role': 'Mage', 'DMG': 250, 'Gold': 1000}
Id : 2
Detail : {'Hero': 'Lancelot', 'Role': 'Assassin', 'DMG': 200, 'Gold': 1000}
Id : 3
Detail : {'Hero': 'Hayabusa', 'Role': 'Assassin', 'DMG': 300, 'Gold': 1000}
Id : 4
Detail : {'Hero': 'Natalia', 'Role': 'Assassin', 'DMG': 150, 'Gold': 1000}
Id : 5
Detail : {'Hero': 'Cecilion', 'Role': 'Mage', 'DMG': 200, 'Gold': 1000}
silahkan masukkan pilihan Id: 3

Your hero is  {'Hero': 'Hayabusa', 'Role': 'Assassin', 'DMG': 300, 'Gold':
1000}
```

Setelah di-run berjalan dengan normal ya... Namun coba perhatikan source codenya. Meski game berjalan dengan baik, tapi source code diatas tidak menggunakan paradigma fungsional. Meskipun tidak mengubah variabel yang ada, tapi fungsinya sangat bergantung pada variabel allheroes. Mari kita buktikan:

```
#hapus dan ubah nama variabel yang dipakai
del allheroes
myheroes = [{"Hero" : "Kagura", "Role" : "Mage", "DMG" : 350, "Gold" : 1000},
         {"Hero" : "Yve", "Role" : "Mage", "DMG" : 250, "Gold" : 1000},
        {"Hero" : "Lancelot", "Role" : "Assassin", "DMG" : 200, "Gold" : 1000},
        {"Hero" : "Hayabusa", "Role" : "Assassin", "DMG" : 300, "Gold" : 1000},
         {"Hero" : "Natalia", "Role" : "Assassin", "DMG" : 150, "Gold" : 1000},
         {"Hero" : "Cecilion", "Role" : "Mage", "DMG" : 200, "Gold" : 1000}]

#dan jalankan game
start()
```

```
Welcome to mini Mobile Legends Game


Select 1 Hero you want to play:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-9c641b3475b1> in <cell line: 11>()
      9
     10 #dan jalankan game
---> 11 start()

 ------------------------------ 1 frames ------------------------------
<ipython-input-4-f52f193c789e> in chooseHero()
      9   print("Select 1 Hero you want to play: ")
     10   selectedhero = {}
---> 11   for i in range(len(allheroes)):
     12     print("Id : {}".format(i))
     13     print("Detail : {}".format(allheroes[i]))

NameError: name 'allheroes' is not defined
```

Game tidak lagi bisa berjalan hanya karena kita ganti nama variabelnya. Hal ini membuktikan bahwa fungsi yang ada sangat bergantung dengan variabel diluar fungsi. Ketergantungan ini sangat berlawanan dengan prinsip fungsional, dimana fungsi seharusnya tidak bergantung pada data diluar fungsi (prinsip pure function). Bagaimana cara memperbaikinya biar jadi fungsi murni(Fungsi yang hasil keluaranya hanya dipengaruhi oleh parameter yang diberikan)? Berarti kita harus memberikan parameter yang dibutuhkan oleh fungsi. Seperti ini:

```
#berikan parameter yang dibutuhkan oleh fungsi
def chooseHero(allheroes):
 print("Select 1 Hero you want to play: ")
 selectedhero = {}
 for i in range(len(allheroes)):
   print("Id : {}".format(i))
   print("Detail : {}".format(allheroes[i]))
 inputs = input("silahkan masukkan pilihan Id: ")
 try:
   selectedhero.update(allheroes[int(inputs)])
   return selectedhero
 except:
   return "Wrong Input!"

#jangan lupa modifikasi isi fungsi start
def start():
 print("Welcome to mini Mobile Legends Game\n\n")
 #untuk disesuaikan dengan modifikasi fungsi
 hero = chooseHero(myheroes)
 print("\nYour hero is ",hero)
 #skenario game berhenti disini lanjutannya belum dibuat

#mari jalankan gamenya
start()
```

```
Welcome to mini Mobile Legends Game

Select 1 Hero you want to play:
Id : 0
Detail : {'Hero': 'Kagura', 'Role': 'Mage', 'DMG': 350, 'Gold': 1000}
Id : 1
Detail : {'Hero': 'Yve', 'Role': 'Mage', 'DMG': 250, 'Gold': 1000}
Id : 2
Detail : {'Hero': 'Lancelot', 'Role': 'Assassin', 'DMG': 200, 'Gold': 1000}
Id : 3
Detail : {'Hero': 'Hayabusa', 'Role': 'Assassin', 'DMG': 300, 'Gold': 1000}
Id : 4
Detail : {'Hero': 'Natalia', 'Role': 'Assassin', 'DMG': 150, 'Gold': 1000}
Id : 5
Detail : {'Hero': 'Cecilion', 'Role': 'Mage', 'DMG': 200, 'Gold': 1000}
silahkan masukkan pilihan Id: 3
```

```
Your hero is  {'Hero': 'Hayabusa', 'Role': 'Assassin', 'DMG': 300, 'Gold':
1000}
```

Voila, game kembali bisa berjalan. Coba bandingkan dengan fungsi chooseHero() sebelumnya!

Lalu, bagaimana cara agar fungsi 'add_stock' menjadi pure function? Sekarang giliran kalian untuk melengkapi kode fungsi dibawah ini agar menjadi pure-function!

Jangan khawatir, sudah ada sedikit petunjuk untuk kalian ikuti:

```
#Untuk membuat pure-function, jangan lupa untuk menghindari penggunaan
variable
# global/data diluar fungsi. Sebagai gantinya,
#1. fungsi harus menerima 'items' sebagai parameter dan
#2. mengembalikan objek 'items' yang telah dimodifikasi
#3. tanpa merubah data luar. Caranya???
#4. bisa dengan membuat salinan objek 'items' untuk dimodifikasi
#5. kemudian dikembalikan sebagai output fungsi
#let's do it....
def add_stock():
```

Untuk penjelasan lebih detail mengenai pure functions, kalian bisa melihat video ini.

2. Lambda Expressions

Lambda expressions adalah fungsi dalam Python yang bersifat anonymous atau tanpa nama. Dalam bahasa yang lebih sederhana, lambda expressions memungkinkan kita untuk membuat fungsi tanpa harus memberikan nama fungsi tersebut. Daripada menggunakan kata kunci def untuk mendefinisikan sebuah fungsi dengan nama tertentu, kita cukup menggunakan kata kunci lambda, diikuti oleh nama variabel atau parameter yang digunakan sebagai input, kemudian menentukan apa yang ingin di-return oleh fungsi tersebut.

Berikut struktur kode penggunaan lambda expression:

```
lambda parameter: expression
```

- lambda : Kata kunci yang menandakan bahwa kita membuat fungsi anonymous lambda.
- parameter : Daftar parameter atau argumen yang digunakan oleh fungsi lambda.
- expression : Ekspresi yang menggambarkan operasi yang ingin dilakukan fungsi lambda.

contoh

```
lambda x : x * 2
```

pada lambda expressions diatas, kita menggunakan x sebagai input, kemudian memproses x tersebut untuk dikalikan dengan 2. Untuk lebih jelasnya kalian bisa simak baris kode berikut:

```
def kali2(input):
 return input * 2
```

```
double = kali2(2)
print(double)
```

```
4
```

```
double = lambda input : input * 2
print(double(2))
```

```
4
```

Sudahkah kalian menemukan perbedaannya? Yaps! menggunakan lambda expression membuat kode menjadi lebih ringkas.

Kedua contoh diatas menampilkan perbedaan antara penggunaan fungsi biasa dan penggunaan lambda expressions. Terlihat lebih simpel menggunakan lambda expressions bukan? :D Dan caranya pun cukup sederhana: Kalian cukup memindahkan parameter dan expresi dalam fungsi sebagai return value.

Lalu bagaimana jika jumlah parameternya lebih dari satu? Lihat contoh perhitungan luas segitiga dibawah ini ya

```
luas_segitiga = lambda alas, tinggi: 0.5 * alas * tinggi
print(luas_segitiga(5,8))
```

```
20.0
```

Dan berikut contoh penggunaan lambda untuk menggantikan fungsi 'calculate_total_stock' pada studi kasus sistem manajemen inventaris barang

```
[ ]  calculate_total_stock = lambda items: sum(item['stock'] for item in items)
```

```
#Tulis kembali kode diatas, dan coba jalankan
```

Selanjutnya, coba ubah fungsi berikut ke dalam lambda expression! Pastikan fungsi lambda yang kalian buat sudah memenuhi kriteria pure function ya…

```
phi = 3.14
def luasLing(r):
 luas = phi*r*r
 return luas
```

3. List Comprehension

Salah fitur pada python yang cukup menarik dan dapat digunakan untuk pemrograman fungsional yang lebih optimal dan ringkas adalah list comprehension.

Kadang dalam beberapa kasus, mengharuskan kita untuk mengisi elemen dari list sebelum melakukan manipulasi data. Di modul 1 kita sudah belajar tentang list dan cara deklarasinya `list = [1, 2, 3, 4, 5 ]`. Nah, untuk membuat daftar list yang lebih panjang, akan merepotkan jika kita harus menuliskan isi datanya satu persatu. Hal tersebut dapat diatasi

dengan menggunakan perulangan dan append list pada setiap iterasinya. Coba perhatikan contoh berikut:

```
list = []
for i in range(40):
   list.append(i)

print(list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, ...
```

Cara tersebut memang tidak salah, namun butuh banyak line code untuk mengisi sebuah list saja, dengan kata lain borosss. Coba bandingkan dengan contoh berikut:

```
list = [i for i in range(40)]
print(list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, ...
```

Nahh kan cukup 1 baris kode untuk membuat list. Kode tersebut merupakan contoh sederhana penerapan list comprehension. Cara ini akan sangat bermanfaat khususnya jika kita perlu membuat list dengan daftar yang panjang tanpa harus menuliskan kode yang panjang pula (yang mana memerlukan banyak waktu dan resource pastinya). Makanya, sudahi cara lama, mari beralih menggunakan list comprehension :D

Tidak hanya itu, kita juga dapat menyisipkan kondisi serta ekspresi lain dalam list comprehension. Perhatikan contoh berikut:

```
#List berisi bilangan kuadrat dari angka genap dalam range 0-10
list = [i*i for i in range(10) if i % 2 == 0 ]
print(list)
```

```
[0, 4, 16, 36, 64]
```

Selain itu, cara tersebut juga bisa diimplementasikan pada string

```
# List comprehension dengan value konstan
names = ["sayang ayang walau gapunya ayang" for x in range(10)]
print(names)
```

```
['sayang ayang walau gapunya ayang', 'sayang ayang walau gapunya ayang' …
```

```
# List comprehension 2D dengan value konstan
names = [["sayang ayang walau gapunya ayang" for x in range(2)]for y in
range(5)]
for name in names:
   print(name)
```

```
['sayang ayang walau gapunya ayang', 'sayang ayang walau gapunya ayang']
['sayang ayang walau gapunya ayang', 'sayang ayang walau gapunya ayang']
['sayang ayang walau gapunya ayang', 'sayang ayang walau gapunya ayang']
['sayang ayang walau gapunya ayang', 'sayang ayang walau gapunya ayang']
```

```
['sayang ayang walau gapunya ayang', 'sayang ayang walau gapunya ayang']
```

```
#List dengan value semua karakter dalam sebuah kalimat/string
sentence = "sayang ayang walau gapunya ayang"
chars = [char for char in sentence]
print(chars)
```

```
['s', 'a', 'y', 'a', 'n', 'g', ' ', 'a', 'y', 'a', 'n', 'g', ' ', 'w', 'a', 'l', ...
```

4. Iterator dan Generator

4.1. Iterator

Sebelum belajar lebih jauh mengenai generator, pertama-tama kita kenalan dulu dengan yang namanya "iterator". Iterator merupakan objek yang berisi jumlah nilai yang dapat dihitung. Iterator adalah objek yang dapat diulangi, sehingga akan mengembalikan data, satu data per satu waktu.

Secara teknis, dalam Python, sebuah iterator adalah objek apa pun yang kelasnya memiliki fungsi next() (__next __() dalam Python 3), fungsi __iter __() yang menghasilkan return self, dan fungsi __init __().

Metode __iter __() digunakan untuk melakukan operasi (menginisialisasi dll), tetapi harus selalu kembali objek iterator sendiri.

Metode __next __() juga digunakan untuk melakukan operasi, dan harus kembali item berikutnya dalam urutan.

Kemudian untuk mencegah iterasi berjalan selamanya, kita dapat menggunakan pernyataan Stop Iteration. Coba perhatikan contoh berikut:

```
class Count:
   # fungsi init untuk proses inisialisasi
   # objek count secara default akan dimulai dari angka 1(start) hingga 5(stop)
   def __init__(self, start=1, stop=5):
       self.num = start
       self.n = stop

   def __iter__(self):
       return self

   def __next__(self):
       num = self.num
       if num > self.n:  # jika num > n/sudah mencapai nilai stop
         raise StopIteration # raise Stop Iteration untuk menghentikan iterasi
       self.num += 1
       return num

# membuat objek iterator dengan parameter (start=5, stop=8)
my_iter = Count(5,8)
```

```
# iterasi secara manual menggunakan method next()
print(next(my_iter))  # counting pertama --> print 5 (sesuai nilai start saat inisialisasi)
print(next(my_iter))  # counting kedua --> print 6
```

```
5
6
```

```
# bisa juga dengan cara berikut:
# dimana next(obj) sama dengan obj.__next__()
print(my_iter.__next__())    # counting ketiga --> print 7
print(my_iter.__next__())    # counting keempat --> print 8 (nilai akhir/stop)
```

```
7
8
```

```
# Berikut ini akan memunculkan error karena item sudah habis
next(my_iter)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-14-4dca00a5e721> in <cell line: 2>()
      1 # Berikut ini akan memunculkan error karena item sudah habis
----> 2 next(my_iter)

<ipython-input-12-4f0fe92a31ae> in __next__(self)
     13         num = self.num
     14         if num > self.n:  # jika num > n/sudah mencapai nilai stop
---> 15           raise StopIteration # raise Stop Iteration untuk menghentikan iterasi
     16         self.num += 1
     17         return num

StopIteration:
```

Karena Iterator adalah objek yang dapat diulangi, maka kita juga bisa menggunakan perulangan untuk mengakses nilai dari suatu iterator. Seperti contoh berikut:

```
# looping objek iterator dengan nilai default/tanpa parameter
for n in Count():
 print(n)
```

```
1
2
3
4
5
```

Suatu objek dikatakan iterable jika dari objek tersebut bisa kita buat iterator. Sebagian besar objek di Python seperti list, tuple, string, dan lain-lain adalah iterable. seperti contoh berikut:

```
# mendefinisikan list
my_list = [5, 4, 7, 3]
print(my_list)
print(type(my_list))
```

```
[5, 4, 7, 3]
```

```
    <class 'list'>
```

```
# membuat iterator dengan iter()
my_iter = iter(my_list)
print(my_iter)
print(type(my_iter))
```

```
    <list_iterator object at 0x7d8a2cebbf40>
    <class 'list_iterator'>
```

Sebenarnya Iterator ini sudah diterapkan di dalam looping for, while dan list comprehension yang sudah kita pelajari sebelumnya, hanya saja tidak tampak secara langsung. Paham kan gais? agak ribet yaa, soalnya iterator itu pake konsep OOP (dari awal bahas class sama objek). Gak fungsional banget kan :D Nahh untuk menghindari hal tersebut, khususnya di pemrograman fungsional, ketika kita menginginkan sebuah iterator, kita membuat sebuah generator.

4.2. Generator

Sederhananya, generator dalam python adalah fungsi yang mengembalikan sebuah objek iterator. Generator adalah cara cepat dan ringkas untuk membuat sebuah iterator.

Secara khusus, generator adalah subtipe dari iterator. Sehingga untuk pemanggilan objek generator dapat dilakukan dengan method **next()** seperti pada iterator.

Generator dibuat dengan memanggil fungsi yang memiliki satu atau lebih ekspresi yield, seperti contoh berikut:

```
# membuat fungsi count yang sama dengan class Count pada materi iterator
# fungsi dengan satu ekspresi yield
def count(start=1, stop=5):
 for i in range(start,stop+1):
   yield i

print(type(count)) # count adalah sebuah fungsi biasa yang mengembalikan nilai
dengan keyword yield, bukan return
```

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

Fungsi count dibuat untuk membuat sebuah iterasi dengan parameter default start = 1 dan stop = 5. Kemudian dibuat perulangan menggunakan for untuk mencetak kembalian dengan keyword yield pada range 1-6 (angka 6 didapatkan dari stop+1 berarti 5+1). Loh kenapa bukan return? kenapa malah yield?

keyword yield mirip dengan keyword return, bedanya yield mengembalikan objek generator. Coba perhatikan kode lanjutan untuk pemanggilan objek generator berikut

```
# membuat generator dengan memanggil fungsi count(start=5, stop=8)
my_gen = count(5,8)    # saat fungsi count dipanggil,
print(type(my_gen))    # dia mengembalikan sebuah generator
```

```
    <class 'generator'>
```

```
# cara mengakses iterasi sama seperti mengakses iterator:
print('counting pertama dengan fungsi next():',next(my_gen))  #  --> print 5
(sesuai nilai start saat inisialisasi)
print('counting kedua dengan method .__next__():',my_gen.__next__() )  #  -->
print 6
print('counting sisany dengan looping for:', end=" ")

for n in my_gen:
 print(n, end=" ")  # a --> print 7 & 8
```

```
counting pertama dengan fungsi next(): 5
counting kedua dengan method .__next__(): 6
counting sisany dengan looping for: 7 8
```

Generator maupun iterator sama sama menggunakan fungsi next ataupun method ._ *next*_() untuk melakukan pemanggilan dan sama sama menyimpan progres iterasi. Namun coba bandingkan dengan kode iterator dalam kelas Count sebelumnya. Dengan tujuan dan output yang sama kita dapat membuat sebuah kode iterator menjadi lebih ringkas dan efisien dengan menggunakan Generator.

Gimana? paham kann gais? Coba kita beralih ke contoh yang lebih sederhana dari penggunaan generator. Perhatikan contoh berikut!

```
def my_generator(n):

   print('Pertama nih boss')
   yield n

   print('Kedua banget')
   yield n + 1

   print('Ketiga')
   yield n + 2

g = my_generator(1)
print(next(g))
```

```
Pertama nih boss
1
```

Loh kok cuma yang pertama aja yang diprint? Inilah kunci dari keyword yield yang kita gunakan.

Tidak seperti keyword return yang biasa kita gunakan untuk membuat fungsi pada umumnya, yang mana akan menghentikan(terminate) function secara keseluruhan.

Yield hanya akan menghentikan sementara(pause) function dan menyimpan semua state variable yang ada didalamnya, sehingga nantinya bisa dilanjutkan kembali. Perhatikan kode berikut

```
print(next(g))
```

```
Kedua banget
2
```

```
print(next(g))
```

```
Ketiga
3
```

Nah Kalau habis iterasinya gimana? yaudah selesaii

```
#Nih StopIteration
print(next(g))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-22-d824875e2a35> in <cell line: 2>()
      1 #Nih StopIteration
----> 2 print(next(g))

StopIteration:
```

Jadi object generator hanya bisa diiterasi untuk sekali saja. Kita harus membuat object generator baru untuk melakukan iterasi kembali. Untuk mengatasi error saat objek generator sudah habis diiterasi, kita bisa menambahkan error handling seperti berikut:

```
try:
 print(next(g))
except:
 print("objek generator telah habis diiterasi")
```

```
objek generator telah habis diiterasi
```

Untuk memahami lebih lanjut mengenai exception handling, kalian bisa baca pada tautan ini

4.3. Generator Expression

Selain itu, generator juga dapat dibuat dengan cara yang lebih ringkas lagi tanpa harus membuat fungsi yang menggunakan keyword yield terlebih dahulu. Kita menyebutnya dengan istilah **generator expression**. Coba perhatikan contoh berikut:

```
count = (i for i in range(5,9))

for i in count:
   print(i, end=" ")
```

```
5 6 7 8
```

kode diatas adalah cara yang sama untuk membuat iterasi dengan range 5-9 yang akan mencetak nilai 5-8
Tidak hanya itu, kita juga dapat menyisipkan kondisi serta ekspresi lain sama seperti saat kita membuat list comprehension. Perhatikan contoh berikut:

```
generator_expression = (i for i in range(11) if i % 2 == 0)
print(generator_expression)
```

```
<generator object <genexpr> at 0x7d8a07f637d0>
```

Namun berbeda dengan list comprehension, kita tidak bisa mencetak nilainya secara langsung menggunakan fungsi print. Kenapa begitu??? karena Generator merupakan implementasi dari **Lazy Evaluation**. Apa lagi ini Lazy Evaluation? :(

Lazy Evaluation merupakan strategi evaluasi yang menunda evaluasi ekspresi hingga nilainya diperlukan dan yang juga menghindari evaluasi berulang. Pada lazy evaluation, dia hanya melakukan evaluasi pada variabel yang dibutuhkan saja, yaah namanya juga malass, ngapain dieval semua kalau tidak dibutuhkan ya kann..

Namun menjadi malas bukan berarti bad attitude dalam pemrograman fungsional, dia justru dapat meningkatkan efisiensi kode dan menghemat banyak sumber daya. Inilah salah satu keunggulan dari paradigma fungsional.

Untuk lebih memahami lazy evaluation, coba kita bandingkan generator dengan list comprehension (yang tidak lazy). cekidot

4.4. Generator vs List Comprehension

Pertama tama, coba perhatikan cara membuat list comprehension dengan generator expression berikut dan temukan apa perbedaannya!

```
list_comp = [i for i in range(10000)]
gen_exp = (i for i in range(10000))
```

Nah dari contoh di atas sudah kelihatan kan bedanya Iyess, kalau List Comprehension dia pakai kurung siku, sedangkan generator pakai kurung biasa, sehingga List Comprehension akan menghasilkan data sequence list dan generator menghasilkan sebuah objek iterator

Selain format penulisan, generator menghasilkan satu item pada satu waktu dan menghasilkan item hanya saat dibutuhkan. Padahal, dalam List Comprehension, Python menyimpan memori untuk seluruh isi list. Dengan demikian dapat dikatakan bahwa ekspresi generator lebih hemat memori daripada List Comprehension. Ga percaya? coba cek kode berikut

```
from sys import getsizeof

#memberikan size untuk list comprehension
x = getsizeof(list_comp)
print("x = ", x)

#memberikan size untuk generator expression
y = getsizeof(gen_exp)
print("y = ", y)
```

```
x =  85176
y =  104
```

Dapat dilihat pada output yang dihasilkan, bahwa generator lebih hemat memori. Tapi apakah hemat waktu eksekusi juga? skuy buktikann..

```
import timeit
 print(timeit.timeit('''list_com = [i for i in range(100) if i % 2 == 0]''',
number=1000000))
```

```
8.253817777999757
```

```
import timeit
 print(timeit.timeit('''gen_exp = [i for i in range(100) if i % 2 == 0]''',
number=1000000))
```

```
0.45977049600060127
```

Iyess, generator menang telak :) jadi dapat disimpulkan bahwa generator lebih hemat memori serta lebih efisien waktu.

LATIHAN PRAKTIKUM

Telah disediakan sebuah struktur data global berupa list **expenses** yang akan digunakan untuk menyimpan daftar pengeluaran harian. Setiap pengeluaran akan direpresentasikan sebagai dictionary dengan atribut **'tanggal', 'deskripsi', dan 'jumlah'**.

```
expenses = [
   {'tanggal': '2023-07-25', 'deskripsi': 'Makan Siang', 'jumlah': 50000},
   {'tanggal': '2023-07-25', 'deskripsi': 'Transportasi', 'jumlah': 25000},
   {'tanggal': '2023-07-26', 'deskripsi': 'Belanja', 'jumlah': 100000},
       ]
```

TODO

1. Buatlah Fungsi **add_expense** untuk menambahkan pengeluaran baru ke dalam expenses. Jangan lupa gunakan **pure-function**
2. Buatlah fungsi **calculate_total_expenses** menggunakan **lambda expression** untuk menghitung total pengeluaran harian.
3. Buatlah fungsi **get_expenses_by_date** menggunakan **list comprehension** untuk menyaring pengeluaran berdasarkan tanggal tertentu.
4. Buatlah fungsi **generate_expenses_report** sebagai **generator** untuk menghasilkan laporan pengeluaran harian dalam bentuk string.
5. Pastikan semua fungsi yang ada sudah menerapkan effect-free programming (pure function), kecuali main
6. Mengubah fungsi yang ada menjadi bentuk lambda

LENGKAPI KODE

```
# TODO 1 Buatlah Fungsi add_expense disini

# TODO 2 Buatlah fungsi calculate_total_expenses disini

# TODO 3 Buatlah fungsi get_expenses_by_date disini

# TODO 4 Buatlah fungsi generate_expenses_report disini

# TODO 5 pastikan semua fungsi yang ada sudah berupa pure function;
```

```
# jika belum, maka modifikasilah

def add_expense_interactively():
    date = input("Masukkan tanggal pengeluaran (YYYY-MM-DD): ")
    description = input("Masukkan deskripsi pengeluaran: ")
    amount = int(input("Masukkan jumlah pengeluaran: "))
    new_expenses = #Panggil fungsi 'add_expense'
    print("Pengeluaran berhasil ditambahkan.")
    return new_expenses

def view_expenses_by_date():
    date = input("Masukkan tanggal (YYYY-MM-DD): ")
    expenses_on_date = #Panggil fungsi 'get_expenses_by_date'
    print(f"\nPengeluaran pada tanggal {date}:")
    for expense in expenses_on_date:
        print(f"{expense['deskripsi']} - Rp {expense['jumlah']}")

def view_expenses_report():
    print("\nLaporan Pengeluaran Harian:")
    expenses_report = #Panggil fungsi 'generate_expenses_report'
    for entry in expenses_report:
        print(entry)

def display_menu():
    print("\n===== Aplikasi Pencatat Pengeluaran Harian =====")
    print("1. Tambah Pengeluaran")
    print("2. Total Pengeluaran Harian")
    print("3. Lihat Pengeluaran berdasarkan Tanggal")
    print("4. Lihat Laporan Pengeluaran Harian")
    print("5. Keluar")

# TODO 6 ubah fungsi berikut ke dalam bentuk lambda
def get_user_input(command):
    return int(input(command))

def main():
    global expenses
    while True:
        display_menu()
        choice = get_user_input("Pilih menu (1/2/3/4/5): ")

        if choice == 1:
            expenses = add_expense_interactively(expenses)
        elif choice == 2:
            total_expenses = calculate_total_expenses(expenses)
            print(f"\nTotal Pengeluaran Harian: Rp {total_expenses}")
        elif choice == 3:
            view_expenses_by_date(expenses)
        elif choice == 4:
            view_expenses_report(expenses)
        elif choice == 5:
             print("Terima kasih telah menggunakan aplikasi kami.")
```

```
            break
        else:
            print("Pilihan tidak valid. Silahkan pilih menu yang benar.")

if __name__ == "__main__":
    main()
```

TUGAS PRAKTIKUM

Pilih salah satu dari dua soal dibawah ini

SOAL A (INTERMEDIATE)

Modifikasi program yang telah kalian kerjakan sebelumnya pada modul 1 dengan menerapkan materi yang sudah dipelajari pada modul 2 dengan ketentuan sebagai berikut:

1. Perbaiki/modifikasi fungsi-fungsi yang kalian buat sebelumnya agar menjadi pure function untuk fungsi yang belum pure
2. Tambahkan fitur 'find' dan 'edit' data
3. Modifikasi fungsi-fungsi di dalam kode kalian dengan menerapkan lambda expression/list comprehension/generator

Semakin banyak materi yang diimplementasikan, maka semakin baik nilai yang didapatkan

SOAL B (ADVANCE)

Kali ini kita akan **bermain** dengan materi yang sudah kita pelajari sebelumnya. Agar kita bisa bermain, kita buat dulu yuk game nya seperti ini:

1. Buat board matrix dengan lebar sesuai dengan inputan dari user. Kalian bisa memanfaatkan **list comprehension** untuk membuat board matrix ini.
2. Terdapat sebuah bidak (simbol A) yang dapat berjalan secara horizontal dan vertikal, serta sebuah goals (simbol O) sebagai target/tujuan permainan.
3. Jika bidak bergerak mencapai posisi tujuan (goals) maka permainan selesai dengan keterangan "menang" dan jika tidak mencapai tujuan maka "kalah".
4. Posisi awal bidak dan posisi goals akan di generate secara acak saat permainan dimulai. Gunakan generator untuk mengenerate position secara random (maksimal 3x). Kalian bisa menambahkan exception handling untuk ini.
5. Jangan lupa sebisa mungkin kalian terapkan juga pure-function dan lambda expression.

Berikut adalah contoh pre-view game yang bisa kalian jadikan inspirasi:

```
~~ Selamat datang dipermainan board game pemrograman fungsional ~~
--------------------------------------------------------------------------------
Anda (A) dapat berjalan secara horizontal dan vertikal untuk menuju target (O)
Gunakan keyboard ASDW untuk bergerak
--------------------------------------------------------------------------------
~~ Selamat bermain ~~

Enter the board width: 5
Enter the board height: 5
```

```
new board generated
- - - - -
- - - - -
- - - - -
A - - - O
- - - - -
New possition (Y/N)? y

new board generated
- - - - -
- - - - -
A - - - -
- - - - -
- - - - O
New possition (Y/N)? n

Let's play... This is your game board
- - - - -
- - - - -
A - - - -
- - - - -
- - - - O
what is your move = ssdddd
- - - - -
- - - - -
- - - - -
- - - - -
- - - - A
You win
```

```
Let's play... This is your game board
- - - - - -
- - - - - -
- - - - - O
- - - - - -
- - - - - -
- - - - - A
- - - - - -
what is your move = aaww
- - - - - -
- - - - - -
- - - - - O
- - - A - -
- - - - - -
- - - - - -
- - - - - -
You lose
Wanna try again?n
See you again...
```

KRITERIA & DETAIL PENILAIAN TUGAS PRAKTIKUM

Kriteria	Soal A	Soal B	Program identik*
1. Program dapat berfungsi	10	20	0
2. Program **mengimplementasikan materi** yang sudah dipelajari pada **modul**	10	20	0
3. Menjelaskan program yang dibuat dengan **baik dan lancar**	20	20	20*
4. Menjelaskan implementasi materi modul dalam program yang dibuat **beserta alasan penggunaannya**	20	20	20*
5. Menjawab pertanyaan dari asisten dengan **baik dan lancar**	20	20	20*
Total Nilai (maksimal)	80	100	60*

*) Jika program identik dengan praktikan lain. Maka akan ada pengurangan nilai pada kedua praktikan