VERSI 2.2

OKTOBER 2025

PEMROGRAMAN FUNGSIONAL

MODUL 2 - Effect-free Programming and Declarative Paradigm

DISUSUN OLEH:

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

ALFI AULIA AZZAHRA

RAHMATUN NIKMAH

TIM LABORATORIUM INFORMATIKA

UNIVERSITAS MUHAMMADIYAH MALANG

PENDAHULUAN

TUJUAN

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

TARGET MODUL

1. Praktikan dapat memahami dan mengimplementasikan konsep pemrograman fungsional basic (effect-free programming) menggunakan bahasa pemrograman Python.
2. Praktikan mampu mengimplementasikan teknik lazy evaluation menggunakan iterator dan generator.
3. Praktikan mampu mengimplementasikan penggunaan exception handling pada sebuah program.
4. Praktikan mampu menentukan teknik yang tepat dalam menyelesaikan masalah pemrograman fungsional perangkat lunak.

PERSIAPAN

1. Laptop/Komputer
2. Python, Google Collab/VS Code/Jupyter Notebook
3. *Source Code Modul 2*

KEYWORDS

Pure Function, Pemrograman Deklaratif, Lazy Evaluation, Generator, Exception Handling

TABLE OF CONTENTS

1. EFFECT-FREE PROGRAMMING

Effect-free programming adalah pendekatan dalam pemrograman yang menekankan pada penulisan kode yang tidak menyebabkan efek samping. Dalam konteks pemrograman fungsional, effect-free programming sangat penting karena mendukung prinsip-prinsip utama dari paradigma ini.

Jika kita ibaratkan dalam konteks matematika, ada sebuah fungsi yang berfungsi sebagai relasi antara dua himpunan yang memetakan setiap elemen dari himpunan pertama (domain) ke element himpunan kedua (range). Nah, fungsi ini biasanya dinotasikan sebagai f(X), dimana X adalah input dan f(X) adalah output.

Contoh Sederhana: $f(x) = 2x^2$

Pada contoh tersebut, jika kita memberikan input x=4, maka output yang akan dihasilkan akan selalu bernilai 32. hal itu disebut dengan deterministik, yang artinya fungsi akan selalu menghasilkan output yang sama untuk input yang sama. Selain itu, fungsi f(x) = $2x^2$ tidak akan mengubah nilai atau variabel lain yang diluar dari perhitungannya. Hasil perhitungannya hanya akan bergantung pada input x dan tidak ada perubahan state yang terjadi. Nah, hal itulah yang disebut dengan Effect-free Programming, dimana fungsi tidak akan mengubah variabel global, tidak melakukan operasi input/output, dan tidak mengubah data di luar ruang lingkupnya.

2. PURE FUNCTION

Jika ingin menciptakan kode yang menerapkan pendekatan Effect-free Programming, salah satu cara yang dapat digunakan adalah Pure Function. ***Pure Function*** sendiri adalah sebuah fungsi yang selalu mengembalikan/me-return output yang sama ketika diberi input yang sama (Seperti contoh matematika diatas). Artinya, untuk setiap inputan yang dilakukan, fungsi akan mengembalikan hasil yang sama secara konsisten, tidak peduli kapan atau dimana fungsi ini dipanggil. Selain itu, pure function tidak akan menyebabkan efek samping apapun selain me-return value. Sehingga, function ini tidak akan memodifikasi state dalam sistem di luar scope-nya. Dan Yup, Pure function itu implementasi dari Effect-free Programming. Perhatikan contoh berikut:

cell code sengaja dipisahkan antara fungsi dan implementasinya. Sebagaimana pemrograman pada umumnya, kita memisahkan antara program utama dan testing. Hal ini bertujuan agar memudahkan kita dalam pengecekan.

Perhatikan code berikut!

```
# nilai atau variabel lain diluar dari perhitungan
y = 10

# fungsi f(x)=2x^2
def f(x):
  return 2*(x*x)

# memberikan input x=4 ke dalam fungsi f(x)
hasil = f(4)
print('f(4)=', hasil)
print('y =', y)

Output:
f(4)= 32
y = 10
```

Pada kode di atas, kita memiliki sebuah fungsi f(x) yang didefinisikan sebagai f(x) = $2x^2$. Fungsi ini menerima sebuah input berupa nilai x, lalu mengembalikan hasil perhitungan 2 * (x * x). Selain itu, terdapat sebuah variabel global bernama y yang bernilai 10, namun variabel ini tidak terlibat dalam proses perhitungan di dalam fungsi.

Ketika kita memanggil f(4), fungsi akan melakukan perhitungan 2 * (4 * 4) sehingga hasilnya adalah 32. Hal ini menunjukkan bahwa output dari fungsi f sepenuhnya bergantung pada nilai input yang diberikan. Tidak peduli berapa kali fungsi tersebut dipanggil, jika inputnya sama yaitu 4, maka hasilnya akan selalu konsisten, yaitu 32.

Karakteristik inilah yang membuat f(x) dapat disebut sebagai pure function. Sebuah pure function memiliki dua ciri utama: pertama, outputnya hanya ditentukan oleh input tanpa dipengaruhi faktor lain; kedua, fungsi tersebut tidak menyebabkan efek samping (side effect) pada variabel atau state di luar ruang lingkupnya. Dalam contoh ini, meskipun terdapat variabel global y, fungsi f tidak mengakses atau mengubah nilainya.

Dengan demikian, pemanggilan print('y =', y) tetap menghasilkan nilai 10, karena variabel global y tidak terpengaruh oleh proses di dalam fungsi. Contoh ini secara sederhana memperlihatkan bagaimana pure function bekerja secara deterministik dan bebas dari perubahan state di luar dirinya.

Dalam contoh lain, kita memiliki fungsi total_nilai yang akan menghitung total nilai yang ada di dalam data input. Fungsi ini hanya bergantung pada parameter yang dibutuhkan, yaitu nilai. Perhatikan contoh berikut:

```
nilai = [
    {'matkul': 'Fungsional', 'nilai': 90},
    {'matkul': 'Web', 'nilai': 65}
]

def total_nilai (nilai):
  total_awal = 0
  for item in nilai:
    total_awal += item['nilai']
  return total_awal

hasil_nilai = total_nilai(nilai)
print("Total nilai: ", hasil_nilai)

Output:
Total nilai:  155
```

Selanjutnya, kita membuat fungsi tambah_nilai guna menambahkan nilai pada salah satu matkul yang ada dalam kumpulan data kita. Fungsi ini akan memodifikasi variabel nilai yang ada di luar fungsi.

```
#deklarasi fungsi
def tambah_nilai(nama_matkul, jumlah_nilai):
    global nilai
    for item in nilai:
        if item['matkul'] == nama_matkul:
            item['nilai'] += jumlah_nilai
            print(f"Nilai {nama_matkul} ditambahkan {jumlah_nilai}. \nTotal
nilai {nama_matkul} saat ini {item['nilai']}")
            break
    else:
        print(f"Mata kuliah {nama_matkul} tidak ditemukan dalam daftar!")

#testing
# Cek nilai awal
print("Nilai awal: ",nilai)

# Menambahkan nilai ke dalam daftar
tambah_nilai('Web', 15)
```

```
# Cek nilai saat ini
print("Nilai update:",nilai)

Output:
Total nilai:  155
```

fungsi tambah_nilai **BUKANLAH** Pure Fuction, karena fugsi ini memodifikasi variabel nilai yang ada di luar fungsi. Selain itu, jika kita memanggil fungsi tambah_nilai beberapa kali, isi dari variabel nilai tersebut akan mengalami perubahan incremental (nilainya naik). Ga Percaya? Coba jalankan code #testing diatas! dapat dilihat adanya penambahan nilai pada matkul Web. dan jika kamu menekannya berkali-kali, maka nilainya juga akan bertambah terus. Itu sebabnya fungsi (tambah_nilai) pada contoh ini tidak termasuk kategori pure fuction.

📌 Latihan 1

```
#testing
# Pemanggilan fungsi murni tambah_nilai
print("Hasil tambah nilai=",tambah_nilai_pure('Web', 15))

# Cek nilai awal
print("Nilai awal=",nilai)

#jika hasil printout keduanya berbeda, maka pure function anda berhasil
```

Untuk penjelasan lebih lanjut mengenai pure function, kalian bisa melihat video ini.

3. FUNGSI DEKLARATIF

Pada Paradigma pemrograman deklaratif, kita ditekankan pada frasa berikut:

```
"Pemrograman deklaratif berfokus pada apa (WHAT) dibandingkan bagaimana (HOW)"
```

Nah, maksud dari pernyataan tersebut adalah: pada pemrograman deklaratif, kita harus mendeklarasikan apa (WHAT) tujuan dan hasil yang ingin kita capai, dibandingkan memikirkan bagaimana (HOW) langkah spesifik untuk mencapainya. Pemrograman Fungsional merupakan salah satu pendekatan pemrograman deklaratif yang populer.

Untuk memahami pemrograman deklaratif lebih lanjut, mari kita bandingkan dengan lawan dari paradigma pemrograman ini, yaitu pemrograman imperatif.

- **Pemrograman imperatif**: Menulis program komputer secara detail bagaimana (HOW) melakukan sesuatu dengan langkah-langkah yang detail dan proses yang jelas. perhatikan potongan kode berikut:

```
numbers = [2, 4, 6, 8, 10]  #variabel data awal
double_numbers = []         #variabel kosong untuk menampung hasil

#iterasi terhadap data
for number in numbers:
    double_numbers.append(number * 2)

#menampilkan hasil
print(numbers)
print(double_numbers)
```

Pada program di atas, kita lihat langkah-langkah yang detail tentang bagaimana (HOW) mencapai hasil, mulai dari menyiapkan variabel kosong double_numbers untuk menyimpan hasil, mendefinisikan loop yang mengiterasi list number, dan menambahkan hasil dari setiap elemennya dikali 2 ke dalam list double_number. Pendekatan ini lebih mendetail tentang bagaimana hasil dicapai, yang merupakan karakteristik dari pemrograman imperatif.

- **Pemrograman Deklaratif**: Mendeskripsikan suatu hal (program) tanpa peduli bagaimana cara mendapatkannya. Cukup fokus pada apa (WHAT) yang harus dicapai tanpa perlu tahu bagaimana cara mencapainya. Contoh:

```
# Mempersiapkan fungsi deklaratif
'''
desc: fungsi double(x) untuk merubah input x menjadi 2x [f(x)=2x]
pre-cond: input x merupakan sembarang data, bisa int, float, dsb
post-cond: 2x sebagai return value
'''

def double(x):
  return x + x
  # Banyak cara untuk mendapatkan 2x. Terserah kita mau pakai rumus apa.
  # Hal ini tidak perlu dijelaskan dalam deskripsi fungsi.

# Memanfaatkan fungsi deklaratif untuk membangun program
double_numbers = map(double, numbers)
print(list(double_numbers))
```

Output:

```
[4, 8, 12, 16, 20]
```

Logika yang sama, bahasa yang sama, dan output yang sama juga, namun Pemrograman Deklaratif lebih ringkas. Pada Program di atas, kita hanya fokus pada apa yang kita inginkan, yaitu memetakan (mapping) setiap elemen dari list numbers menjadi 2 kalinya dengan menggunakan fungsi map dan double. Disini kita tidak mendetailkan langkah-langkah iterasi secara eksplisit untuk mencapai hasil ini. Kita cukup tau apa (WHAT) yang dilakukan oleh fungsi map dan double, tanpa perlu tau bagaimana (HOW) cara mereka melakukannya. Lebih detail tentang fungsi map akan dibahas pada modul selanjutnya.

Begitu pula dengan fungsi double yang sudah kita deklarasikan di awal. Setiap fungsi yang dibangun dalam paradigma fungsional harus dilengkapi dengan deskripsi fungsi, input apa yang dibutuhkan, dan bentuk output yang dihasilkan. Hal ini membuat kode lebih mudah dibaca dan dipahami, serta lebih deklaratif.

4. ITERATOR DAN GENERATOR

4.1. TENTANG PARADIGMA FUNGSIONAL

Suatu proses yang diulang lebih dari sekali dengan menerapkan logika yang sama disebut dengan iterasi. Dalam pemrograman—termasuk python, penggunaan loop—seperti for atau while—yang dibuat dengan kondisi tertentu untuk mengulang suatu proses yang sama hingga kondisi tersebut terpenuhi, juga dikenal sebagai iterasi.

Sedangkan Iterator adalah objek yang memungkinkan kita untuk melintasi elemen-elemen dari suatu koleksi data, seperti list, tuple, atau set, satu per satu. Dengan kata lain, iterator bertugas untuk mengambil nilai berikutnya dari sebuah objek yang bisa diiterasi.

- **Iterasi** itu seperti sedang menelusuri atau mengulangi suatu proses yang sama untuk setiap elemen dalam suatu koleksi data secara berurutan.
- **Objek yang dapat diiterasi (iterable object)** adalah wadah atau koleksi data yang elemen-elemennya bisa kita telusuri, seperti `list` atau `string`.
- **Iterator** adalah alat yang secara spesifik melakukan proses penelusuran tersebut. Ia memiliki "ingatan" di mana ia terakhir kali berada, sehingga bisa mengambil elemen selanjutnya.

Iterator bekerja dengan konsep **lazy evaluation** (evaluasi yang "malas"), yang artinya ia hanya akan bekerja—menghitung dan mengambil nilai— ketika nilai tersebut benar-benar dibutuhkan. Hal ini membantu kita menghemat memori dan menghindari proses perhitungan yang tidak perlu, karena iterator tidak menghitung semua nilai sekaligus. Ia hanya akan mengambil satu per satu, sesuai dengan permintaan.

Iterator diimplementasikan menggunakan class dan variabel lokal untuk iterasi. Dalam menggunakan iterator, dibutuhkan sebuah class yang berisikan:

- Fungsi `__init__()` sebagai **constructor**, tempat kita menginisialisasikan nilai awal seperti limit. Jika tidak dibutuhkan inisialisasi nilai awal dalam iterator, maka fungsi ini boleh ditiadakan.
- Fungsi `__iter__()` yang digunakan untuk membuat sebuah iterator dengan sebuah iterable object (self). Fungsi ini juga digunakan untuk melakukan operasi (menginisiasi dan lain-lain), tetapi harus selalu kembali ke objek iterator itu sendiri (return self).
- Fungsi `__next__()` digunakan untuk memanggil elemen selanjutnya dalam iterable object.

seperti ini contohnya:

```
# Membuat class iterator untuk menghasilkan bilangan kelipatan 3 hingga
batas tertentu
class KelipatanTiga:
    def __init__(self, limit):
        # Menyimpan nilai batas maksimum
        self.limit = limit
        # Nilai awal iterasi dimulai dari 0
        self.current = 0

    def __iter__(self):
        # Mengembalikan objek itu sendiri sebagai iterator
        return self

    def __next__(self):
        # Mengecek apakah nilai saat ini masih dalam batas
        if self.current <= self.limit:
            # Simpan nilai yang akan dikembalikan
            result = self.current
            # Tambahkan 3 untuk iterasi berikutnya
            self.current += 3
            return result
        else:
            # Jika sudah melewati batas, hentikan iterasi
            raise StopIteration
```

Dengan demikian, apakah Iterator bisa kita kategorikan menggunakan konsep OOP (Object Oriented Programming) karena diimplementasikan menggunakan class? *Iterator sendiri adalah konsep umum yang bisa digunakan dalam berbagai paradigma pemrograman, tetapi cara implementasinya bisa berbeda tergantung paradigma yang digunakan.*

Dalam paradigma Fungsional, penggunaan class umumnya dihindari karena fokus utamanya adalah pada fungsi sebagai komponen utama. Selain konsep class lebih identik dengan paradigma OOP, membuat iterator dengan cara seperti diatas sangatlah melelahkan dan kurang efisien. Huffft..

Untungnya, Python menyediakan cara yang lebih sederhana: hampir semua objek Python (seperti list, tuple, string, dan sebagainya) adalah iterable, dan kita bisa langsung membuat iterator dari objek-objek tersebut menggunakan fungsi bawaan `iter()`.

seperti berikut contoh implementasinya:

```
# Mendefinisikan list berisi nama-nama hari
list_hari = ['Senin', 'Selasa', 'Rabu', 'Kamis', 'Jumat']
print(list_hari)       # Menampilkan isi list
print(type(list_hari)) # Menampilkan tipe data list_hari (yaitu list)
print()
```

```
['Senin', 'Selasa', 'Rabu', 'Kamis', 'Jumat']
<class 'list'>
```

```
# Membuat iterator dari list menggunakan fungsi bawaan iter()
iterator_hari = iter(list_hari)
print(iterator_hari)       # Menampilkan objek iterator
print(type(iterator_hari)) # Menampilkan tipe objek iterator (yaitu <class 'list_itei
```

```
<list_iterator object at 0x79546653d2a0>
<class 'list_iterator'>
```

```
# gunakan fungsi next() untuk menampilkan data
next(iterator_hari)
```

```
'Selasa'
```

Contoh ini menunjukkan bahwa meskipun `list_hari` adalah **iterable**, ia bukan iterator. Tetapi dengan melakukan casting menggunakan `iter()`, kita bisa mengubahnya menjadi objek iterator yang lebih sesuai dengan pendekatan fungsional.

Jika contoh barusan hanya memanfaatkan casting data--yang mana harus ada data nya dulu untuk di casting-- bagaimana jika kita perlu membangun sebuah iterator sendiri dari nol?! Apakah harus membuat class seperti sebelumnya? Ataukah ada pendekatan lain yang lebih fungsional. Tentu ada! Oleh sebab itu, Python menyediakan alternatif lain yang lebih ringkas, lebih idiomatis, dan lebih sesuai dengan paradigma fungsional, yaitu Generator.

Jika ingin menggunakan pendekatan OOP dengan class maupun fungsional dengan `iter()`, sebenarnya keduanya sah-sah saja digunakan. Namun, untuk kasus

sederhana, pendekatan tersebut terkadang terasa terlalu panjang dan kurang praktis. Karena itulah Python memperkenalkan konsep **Generator**—yaitu fungsi khusus yang dapat menghasilkan iterator dengan sintaks lebih ringkas serta lebih sesuai dengan paradigma fungsional.

4.2. GENERATOR

Perhatikan kode dibawah ini:

```
# Membuat generator untuk menghasilkan bilangan kelipatan 3 hingga batas tertentu
def kelipatan_tiga_generator(limit):
    current = 0
    while current <= limit:
        yield current
        current += 3

# Menggunakan generator
generate = kelipatan_tiga_generator(15)
print(generate)

# Menampilkan isinya
for angka in generate:
    print(angka, end='-')

#PERCOBAAN 3
```

```
<generator object kelipatan_tiga_generator at 0x7954679d6080>
0-3-6-9-12-15-
```

Dengan generator, kita tidak perlu lagi menulis class, `__iter__()`, atau `__next__()`. Cukup menggunakan kata kunci yield di dalam fungsi, maka Python secara otomatis akan menangani proses iterasi di balik layar.

Generator adalah jenis khusus dari fungsi pada Python yang menghasilkan nilai secara malas atau ***lazy evaluation*** dan secara iteratif, daripada mengembalikan satu nilai secara langsung seperti fungsi biasa.

Generator menggunakan kata kunci `yield` untuk mengembalikan nilai, yang membuatnya sangat berguna untuk menghasilkan urutan nilai dalam suatu konteks dimana perlu menghemat memori atau menangani data secara berurutan. Untuk pemanggilan objek generator dapat dilakukan dengan method `next()` seperti pada iterator. Perhatikan contoh dibawah ini:

Pertama, kita perlu membuat sebuah fungsi generator dengan menambahkan keyword `yield` sebagai pengganti keyword `return`. Jangan lupa untuk menambahkan deklarasi/deksripsi pada setiap fungsi yang kita bangun yaa, agar lebih mudah dipahami APA tujuan dari fungsi tersebut!

```
"""
desc: fungsi generator untuk menghasilkan bilangan kelipatan 3 dari 0
hingga
      'limit' tertentu.
pre-cond: 'limit' harus berupa int >= 0
post-cond: object generator berisi bilangan kelipatan 3 dari 0 sampai
'limit'
"""
# Definisi fungsi generator
def kelipatan_tiga(limit):
    n = 0
    while n <= limit:
        yield n          # Mengembalikan nilai n
        n += 3           # n Tambah 3 untuk setiap iterasi
```

Pada kode diatas, fungsi `kelipatan_tiga(limit)` adalah fungsi generator yang menghasilkan object generator yang berisikan bilangan kelipatan 3 dari 0 sampai `limit`. Keyword yield digunakan untuk mengembalikan nilai secara bertahap setiap kali data dari object generator dipanggil tanpa menghentikan eksekusi fungsi sepenuhnya.

Selanjutnya, kita siapkan sebuah variabel generator_tiga sebagai keranjang penyimpanan hasil dari fungsi `kelipatan_tiga`. `generator_tiga` adalah objek generator yang dihasilkan dengan memanggil fungsi `kelipatan_tiga(15)`. kalian bisa membedakan keduanya dengan cara memanggil fungsi `print()` dan `type` seperti ini:

```
# Membuat objek generator
generator_tiga = kelipatan_tiga(15)

# Menampilkan tipe objek
print(type(kelipatan_tiga))
print(type(generator_tiga))
print(kelipatan_tiga)           # Alamat fungsi
print(generator_tiga)           # Alamat objek generator

Output:
<class 'function'>
<class 'generator'>
<function kelipatan_tiga at 0x7954665a1d00>
<generator object kelipatan_tiga at 0x7954679d5f00>
```

Objek generator tidak dapat dicetak secara langsung karena dia lazy. Kita perlu melakukan iterasi/loop untuk mendapatkan isinya satu persatu atau menggunakan method next() seperti berikut:

```
print("Mengambil dua nilai pertama dengan next()")
print(next(generator_tiga)) #nilai pertama 0
print(next(generator_tiga)) #nilai kedua 3
print()

print("Melanjutkan hasil dari generator menggunakan for-loop")
for num in generator_tiga:
    print(num, end=' ') #nilai selanjutnya dipisah dengan spasi

print('\n')
print("Setelah generator habis(mencapai limit), pemanggilan next() akan error")
print(next(generator_tiga)) #StopIteration error
```

```
Mengambil dua nilai pertama dengan next()
0
3

Melanjutkan hasil dari generator menggunakan for-loop
6 9 12 15

Setelah generator habis(mencapai limit), pemanggilan next() akan error
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
/tmp/ipython-input-3816615373.py in <cell line: 0>()
     10 print('\n')
     11 print("Setelah generator habis(mencapai limit), pemanggilan next() akan error")
---> 12 print(next(generator_tiga))     # StopIteration error

StopIteration:
```

Disini kita menggunakan fungsi `next(generator_tiga)` untuk mengambil nilai berikutnya dari generator secara berurutan. Loop `for num in generator_tiga` juga dapat digunakan untuk mencetak setiap bilangan ganjil selanjutnya yang akan dihasilkan oleh generator.

Setelah program dieksekusi, maka akan dihasilkan angka kelipatan tiga kurang dari limit (15) yaitu 0, 3, 6, 9, 12, dan 15. Perintah `next()` berikutnya akan memunculkan exception `StopIteration`, yang mana artinya elemen berikutnya dari generator tersebut telah habis dan tidak ada lagi nilai yang bisa dihasilkan. Objek generator juga tidak bisa digunakan lagi. Kalian harus membuat objek generator baru jika ingin mengulangi prosesnya.

4.3. GENERATOR EXPRESSION

Generator dapat dibuat dengan cara yang lebih simple tanpa harus membuat fungsi yang menggunakan kata kunci `yield` terlebih dahulu. Hal ini biasanya disebut dengan **Generator Expression**. Cara membuat nya pun hanya perlu menggunakan tanda kurung dan perulangan didalamnya (bisa juga ditambahkan kondisi `if`). Perhatikan contoh berikut:

```
# Generator expression untuk angka kelipatan 3
gen_expr = (i for i in range(16) if i % 3 == 0)

# Menampilkan objek generator expression
print(gen_expr)

# Menampilkan isi dari objek generator menggunakan iterasi for
for val in gen_expr:
    print(val, end='-')

Output:
<generator object <genexpr> at 0x79546644d490>
0-3-6-9-12-15-
```

Kode diatas adalah cara yang sama untuk membuat iterasi dengan range 16 yang akan mencetak angka kelipatan 3 dari 0 sampai 15. Lebih sederhana lagi daripada harus membuat fungsi generator yang panjang seperti sebelumnya ataupun class iterator yang rumit kan... Nah ini lah fungsional programing yang deklaratif guys!

📌 Latihan 3

```
Bandingkan kode program beserta outputnya antara fungsi
kelipatan_tiga_generator(limit) pada materi 4.2 Generator dengan #Generator
expression untuk angka kelipatan 3 barusan!
```

5. LAZY EVALUATION

Pada pembahasan sebelumnya, istilah ini sudah sering disebutkan. Tapi maksudnya apa sih? Coba perhatikan Contoh ini:

```
ini_range = range(1, 100)
ini_list = list(range(1, 100))

#PERCOBAAN 4a
```

Kode tersebut merupakan contoh penggunaan range dan list pada Python. Memang kelihatan mirip, tetapi nyatanya ada sebuah perbedaan anata kedua kode terasebut. Pada range, kita menggunakan konsep lazy evaluation, yang mana evaluasi ekspresi akan menghasikan satu item pada satu waktu dan menghasilkan item hanya pada saat dibutuhkan. Nah, item dari objek yang lazy tersebut tidak bisa diakses secara langsung. Sehingga kita menbutuhkan iterasi/loop untuk dapat mengaksesnya satu per satu. Untuk perbedaan output silahkan lihat code dibawah ini:

```
print("data range = ",ini_range)
print("data list = ",ini_list)

Output:
data range =  range(1, 100)
data list =  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93,
94, 95, 96, 97, 98, 99]
```

Jadi, kita bisa beranggapan bahwa menggunakan range dan lazy dapat menghemat memori daripada kita menggunakan list, mengingat data yang tidak bisa diakses yang artinya data tersebut belum tersedia di memori internal. Mari kita lihat hasil output kode ini:

```
from sys import getsizeof
x = getsizeof(ini_range)
print("Size dari range: ", x)

y = getsizeof(ini_list)
print("Size dari list:", y)

Output:
Size dari range:  48
Size dari list: 856

#PERCOBAAN 4b
```

Seperti yang kalian lihat, size range lebih sedikit daripada list. Hal ini dapat membuktikan bahwa range dapat menghemat memori daripada list. Hal ini juga membuktikan bahwa sesuatu yang malas tidak selalu berkonotasi negatif, aplagi pada

pemrograman fungsional. Yang disebut Lazy disini malah dapat bekerja dengan lebih cepat dan lebih hemat. Konsep ini merupakan salah satu teknik unggulan pada paradigma fungsional.

📌 Latihan 4

```
Coba kalian modifikasi jumlah nilai dari data yang ada (range dan list)
menjadi lebih besar/banyak dan perhatikan bedanya dalam hal memory size!
```

6. EXCEPTION HANDLING

Kalian pasti sangat familiar dengan istilah ini kan? Exception handling merupakan sebuah mekanisme untuk menangani suatu program agar jika terjadi error, program tersebut tidak akan crash atau berhenti dengan sendirinya. Exception handling biasanya dipakai jika kita ingin mengidentifikasi, menangani, dan memberikan respon spesifik kepada suatu kesalahan tanpa menghentikan program tersebut secara tiba-tiba.

Cara kerjanya bisa kita ibaratkan seperti pintu darurat pada sebuah gedung. Jika terjadi kebakaran/bencana yang tidak diinginkan (jika dalam program itu seperti error), kita bisa langsung pergi berlari ke tangga darurat yang sudah disediakan dan diberitahukan pada gedung tersebut agar dapat tetap aman dan tertangani.

Begitu juga dengan exception handling, ketika ada kesalahan saat program berjalan, daripada program langsung berhenti, kita dapat mengarahkan program untuk menampilkan pesan kesalahan yang sesuai, menyimpan log, atau melakukan tindakan lain yang sudah disiapkan agar program tetap berjalan.

Biasanya Exception handling menggunakan syntax berikut:

```
try:
      # Potongan kode yang mungkin akan menghasilkan error/exception
except SomeException:
      # Blok yang dieksekusi jika exception terjadi
else:
     # Blok yang dieksekusi jika tidak ada exception yang terjadi
finally:
    # Kode di blok ini akan selalu dieksekusi terlepas dari adanya
exception atau tidak
```

- `Try`: pada blok ini kita dapat menempatkan potongan kode yang mungkin menghasilkan exception,kemudian akan menangani exception tersebut dengan blok except yang sesuai.
- `Except`: blok ini digunakan untuk menangani exception atau kesalahan yang mungkin akan terjadi di blok try. blok ini tebagi lagi menjadi dua, yaitu `jenisException` sebagai jenis atau kelas exception yang sedang ditangani. Lalu

yang kedua adalah variabel_exception sebagai variabel yang dapat digunakan untuk mengakses informasi tambahan tentang exception yang terjadi, variabel ini bersifat opsional tergantung dengan kebutuhan.

- `Else`: blok ini digunakan untuk menentukan kode yang akan dieksekusi jika tidak ada exception yang terjadi dalam blok try.
- `Finally`: Terakhir adalah blok Finally, dimana blok ini akan mengeksekusi kode didalamnya, baik jika terjadi exception maupun tidak. Blok ini juga mengindikasikan untuk melakukan tindakan final, seperti membersihkan sumber daya atau menutup koneksi, tanpa peduli bagaimana alur eksekusi program.

Masih bingung? Mari kita Perhatikan contoh dibawah ini:

```
a = 10
b = 0

try:
  hasil = a / b
except ZeroDivisionError:
  print("Tidak bisa membagi dengan nol")
else:
  print("Hasil pembagian:", hasil)
finally:
  print("Program selesai")

Output:
Tidak bisa membagi dengan nol
Program selesai
```

Seperti yang terlihat pada kode, program mencoba melakukan pembagian 10 / 0. Karena pembagian dengan nol tidak diperbolehkan dalam Python, maka akan terjadi error bertipe ZeroDivisionError. Ketika error ini muncul, program langsung masuk ke blok except dan menampilkan pesan: "Tidak bisa membagi dengan nol". Setelah itu, terlepas dari apakah terjadi error atau tidak, program akan selalu mengeksekusi blok finally, yang dalam hal ini mencetak "Program selesai". Blok else tidak dijalankan karena hanya akan dieksekusi jika tidak terjadi error saat pembagian. Coba kalian ganti value dari b (menjadi selain 0) lalu run kembali programnya!

Tapi bagaimana kalau kita ingin punya banyak penanganan (lebih dari satu exception)? Tentu bisa dong, coba perhatikan kode dibawah ini:

```
data = ["10", "0", "abc", None]

for nilai in data:
```

```
    try:
        print(f"Memproses nilai: {nilai}")
        angka = int(nilai)  # Konversi ke integer (bisa menyebabkan
ValueError / TypeError)
        hasil = 100 / angka # Pembagian (bisa menyebabkan
ZeroDivisionError)
    except ValueError:
        print("Terjadi ValueError: Nilai tidak bisa dikonversi ke
integer.")
    except ZeroDivisionError:
        print("Terjadi ZeroDivisionError: Tidak bisa membagi dengan nol.")
    except TypeError:
        print("Terjadi TypeError: Tipe data tidak valid.")
    except Exception as e:
        print(f"Terjadi kesalahan lain: {e}")
    else:
        print(f"Hasil pembagian 100 / {angka} = {hasil}")
    finally:
        print("Satu proses selesai.\n")

print("Semua proses selesai")

Output:
Memproses nilai: 10
Hasil pembagian 100 / 10 = 10.0
Satu proses selesai.

Memproses nilai: 0
Terjadi ZeroDivisionError: Tidak bisa membagi dengan nol.
Satu proses selesai.

Memproses nilai: abc
Terjadi ValueError: Nilai tidak bisa dikonversi ke integer.
Satu proses selesai.

Memproses nilai: None
Terjadi TypeError: Tipe data tidak valid.
Satu proses selesai.

Semua proses selesai

# PERCOBAAN 5
```

Dalam contoh ini, kita memiliki list berisi beberapa nilai: `"10"`, `"0"`, `"abc"`, dan `None`. Setiap nilai akan diproses satu per satu dalam sebuah perulangan `for`. Di dalam blok `try`, program mencoba mengkonversi nilai menjadi integer dan kemudian melakukan pembagian 100 dengan nilai tersebut. Jika nilai tidak bisa dikonversi ke integer, seperti `"abc"` atau `None`, maka akan muncul `ValueError` atau `TypeError`, dan pesan kesalahan yang sesuai akan ditampilkan. Jika nilai berhasil dikonversi namun bernilai nol, maka saat dilakukan pembagian akan memunculkan `ZeroDivisionError`. Semua kemungkinan kesalahan tersebut ditangani dengan `except` yang sesuai, sehingga program tidak akan berhenti secara tiba-tiba. Jika tidak terjadi kesalahan, maka hasil pembagian akan ditampilkan oleh blok `else`. Terakhir, blok `finally` selalu dijalankan untuk menandai akhir dari setiap proses, apapun hasilnya. Pendekatan ini menunjukkan betapa pentingnya exception handling untuk menjaga program tetap berjalan meskipun terdapat data yang bermasalah.

📌 Latihan 5

```
Pada blok exception handling, pindahkan blok except Exception as e: menjadi
blok except pertama (tepat setelah baris try). Apakah praktik seperti ini
ideal?
```

CODELAB

CODELAB dikerjakan dalam bentuk laporan dengan merujuk pada laporan ini.

CODELAB 1

Kamu mendapatkan sebuah program, dimana Program tersebut digunakan untuk menganalisis data retail. Akan tetapi, fungsi tersebut masih belum jelas digunakan untuk apa dan masih ada yang tidak lengkap. Tugasmu adalah membuat deskripsi dari fungsi tersebut agar diketahui penggunaannya dan mengisi kode yang belum lengkap!

```
data = {
    "produk": ["Sabun", "Sampo", "Pasta Gigi", "Hand Sanitizer"],
    "harga": [12000, 15000, 10000, 20000],
    "stok": [10, 0, 5, 2]
}


'''
DESKRIPSI FUNGSI DISINI
'''
def cari_produk_terendah_by(dataProduk, dataRendah):
  terendah = min(dataRendah)
  idxTerendah = dataRendah.index(terendah)
  return dataProduk[idxTerendah]


'''
DESKRIPSI FUNGSI DISINI
'''
def total_stok(data):
  # menghitung total stok menggunakan generator



'''
DESKRIPSI FUNGSI DISINI
'''
def cari_data_by(data,key):
  # Hitung harga rata-rata menggunakan exception handling jika key tidak
ditemukan
```

```
'''
DESKRIPSI FUNGSI DISINI
'''
def hitung_penghasilan(data1,data2):
  # menggunakan pure function untuk menghitung penghasilan setiap barang
  # dengan mengalikan dua data

#hasil dari data harga dan stok


# Tampilkan data dengan menambahkan hasil dari hitung_penghasilan


Contoh output dapat dilihat disini.
```

CODELAB 2

Kode berikut masih memiliki error yang membuat programnya terhenti. Identifikasi dan perbaiki kode mana yang berpotensi menyebabkan error, perbaiki kode tersebut dengan menambahkan Exception Handling yang sesuai!
Pastikan bahwa seluruh fungsi bersifat pure dan tidak mencetak langsung dari dalam fungsi.

```
'''
DESKRIPSI FUNGSI DISINI
'''
def bagi(a, b):
    return a / b

'''
DESKRIPSI FUNGSI DISINI
'''
def ambil_produk_ke(data, index):
    return data[index]

'''
DESKRIPSI FUNGSI DISINI
'''
def generator_kelipatan(n):
    for i in range(n):
        if i % 3 == 0:
            yield i
```

```
# Data
produk = ['Sabun', 'Sampo']
harga = [12000, 15000]

print(bagi(10, 0))
print(ambil_produk_ke(produk, 5))
print(list(generator_kelipatan('lima')))

Contoh output yang benar:
2.0
Sampo
[0, 3, 6, 9, 12]
```

TUGAS

TUGAS 1

Melanjutkan tugas modul 1 agar memenuhi paradigma fungsional. Untuk melihat tema silahkan mengakses link ini.

Pada modul sebelumnya, kalian telah membuat beberapa fungsi CRUD untuk menyelesaikan tugas praktikum modul 1. Pada modul 2 kali ini, tugas kalian adalah mengubah **SELURUH** fungsi yang telah dibuat pada tugas modul 1 agar memenuhi paradigma fungsional, yaitu menjadi fungsi murni (**pure function**) dan **deklaratif**. Kalian harus bisa menunjukkan dan menjelaskan proses modifikasi yang kalian lakukan (**before-after code modification**) pada asisten.

TUGAS 2

Telah disediakan sebuah data yang perlu diolah dengan menggunakan konsep lazy-functional. Anda diminta untuk menyelesaikan ketiga masalah di bawah ini dengan memanfaatkan generator.

Ketentuan Pengerjaan:

- Anda **wajib** menggunakan kedua jenis implementasi generator: fungsi **generator** (fungsi yang menggunakan `yield`) dan **generator expression**.
- Dari tiga soal yang diberikan, minimal harus ada **satu** soal yang diselesaikan menggunakan fungsi generator dan minimal satu soal yang diselesaikan menggunakan generator expression.
- Anda dibebaskan untuk memilih soal mana yang akan dikerjakan dengan fungsi generator ataupun generator expression, selama kedua ketentuan di atas terpenuhi. (Contoh: 1 fungsi generator & 2 generator expression, atau 2 fungsi generator & 1 generator expression).
- Untuk menampilkan hasil, Anda wajib mendemonstrasikan cara kerja generator secara manual. Gunakan perulangan `while` dan `next()` untuk mengambil setiap nilai, serta terapkan `try-except StopIteration` untuk menangani kondisi ketika generator telah habis dan menghentikan perulangan, dan tambahkan print untuk mengetahui apakah iterator sudah selesai. Dilarang menggunakan perulangan for untuk menampilkan hasil akhir.

NIM GENAP

Kamu diberikan sebuah data karyawan untuk diolah secara fungsional:

```
karyawan = [
    {'nama': 'Zaky', 'gaji': 5000000, 'bonus': 1000000, 'status_aktif': True},
    {'nama': 'Fitra', 'gaji': 4500000, 'bonus': None, 'status_aktif': True},
    {'nama': 'Fia', 'gaji': 5200000, 'bonus': 500000, 'status_aktif': True},
```

```
    {'nama': 'Adit', 'gaji': 6000000, 'bonus': 1000000, 'status_aktif': False},
    {'nama': 'Faizal', 'gaji': 4000000, 'bonus': 700000, 'status_aktif': True},
    {'nama': 'Radan', 'gaji': 5500000, 'bonus': 'tidak ada', 'status_aktif': True},
    {'nama': 'Wempy', 'gaji': 4800000, 'bonus': 600000, 'status_aktif': True},
    {'nama': 'Alfi', 'gaji': 5000000, 'bonus': 800000, 'status_aktif': False},
    {'nama': 'Hakim', 'gaji': 5200000, 'bonus': 1000000, 'status_aktif': True},
    {'nama': 'Rama', 'gaji': None, 'bonus': 750000, 'status_aktif': True}
]
```

1. Buatlah generator generator/generator expression yang hanya menghasilkan data karyawan dengan `status_aktif == True.`
2. Buatlah sebuah generator/generator expression yang menghasilkan nama karyawan dengan data yang tidak valid (gaji atau bonus bukan integer).
3. Buatlah sebuah generator yang menghitung kontribusi tiap karyawan aktif dengan data valid terhadap total kompensasi (gaji + bonus). Hasil generator harus berupa dictionary dengan key:
   a. "nama" sebagai nama karyawan
   b. "kompensasi" sebagai total gaji ditambah bonus
   c. "kontribusi" sebagai persentase kontribusi terhadap total kompensasi semua karyawan valid (format string xx.xx%)

Contoh output dapat dilihat disini.

NIM GANJIL

Kamu telah diberikan sebuah data sebagai berikut:

```
data_angka = [
    {"sensor": "S1", "nilai": 0},
    {"sensor": "S2", "nilai": 15},
    {"sensor": "S3", "nilai": -3},
    {"sensor": "S4", "nilai": 20},
    {"sensor": "S5", "nilai": "error"},
    {"sensor": "S6", "nilai": 8},
    {"sensor": "S7", "nilai": None},
    {"sensor": "S8", "nilai": 33},
    {"sensor": "S9", "nilai": 5},
    {"sensor": "S10", "nilai": 73},
    {"sensor": "S11", "nilai": -17},
    {"sensor": "S12", "nilai": 100},
    {"sensor": "S13", "nilai": 60},
    {"sensor": "S14", "nilai": -7},
    {"sensor": "S15", "nilai": 90},
    {"sensor": "S16", "nilai": "invalid"},
    {"sensor": "S17", "nilai": 30},
    {"sensor": "S18", "nilai": -45},
```

```
    {"sensor": "S19", "nilai": 88},
    {"sensor": "S20", "nilai": 15}
]
```

Data ini mensimulasikan kumpulan angka yang dikumpulkan dari berbagai sensor digital, namun tidak semuanya valid untuk diproses langsung. Oleh karena itu, kamu diminta mengimplementasikan solusi pemrosesan data secara fungsional.

1. Buatlah sebuah generator/generator expression yang hanya menghasilkan ID sensor dari semua sensor yang memiliki nilai bilangan bulat positif.
2. Buatlah sebuah generator yang memproses nilai dari data yang valid (bilangan bulat positif), lalu menghasilkan hasil dari pembagian 1000 / nilai.
3. Buatlah sebuah generator/generator expression yang menghitung kontribusi tiap sensor valid (bilangan bulat positif) terhadap total semua nilai sensor valid. Hasil generator harus berupa dictionary dengan key:
   a. "sensor" sebagai id sensor
   b. "nilai" sebagai nilai asli sensor
   c. "kontribusi" sebagai persentase kontribusi terhadap total nilai sensor valid (format string xx.xx%)

Contoh output dapat dilihat disini.

RUBRIK PENILAIAN

Komponen Penilaian	Bobot (%)
Codelab	**15**
Tugas 1	
Kode / Kelengkapan Fitur	15
Originalitas kode	10
Pemahaman	20
Tugas 2	
Kode / Kelengkapan Fitur	15
Originalitas kode	5
Pemahaman	20
Total Akhir	**100**