VERSION 2.1

31 Agustus 2024

[PEMROGRAMAN FUNGSIONAL]

Modul 2 – Functional Programming as Effect-free Programming and Declarative Paradigm

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

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. 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 menentukan teknik yang tepat dalam menyelesaikan masalah pemrograman fungsional perangkat lunak.

PERSIAPAN SOFTWARE/APLIKASI

- Komputer/Laptop
- Sistem operasi Windows/Linux/Mac OS/Android
- IDE Pycharm/Google Collab/ Jupyter Notebook
- Source Code Google Colab Modul 2

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.

Dalam konsep matematika, fungsi adalah relasi antara dua himpunan yang memetakan setiap elemen dari himpunan pertama (domain) ke elemen himpunan kedua (range). Fungsi biasanya dinotasikan sebagai $f(x)$, di mana x adalah input dan $f(x)$ adalah output.

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

Dalam fungsi ini, jika kita memberi input $x = 3$, maka outputnya akan selalu 9. Hal ini disebut **deterministik**, yaitu fungsi selalu menghasilkan output yang sama untuk input yang sama. Selain itu, fungsi matematika $f(x) = x^2$ tidak mengubah nilai atau variabel lain di luar perhitungannya. Hasil perhitungannya hanya akan bergantung pada input x dan tidak ada perubahan state yang terjadi. Hal ini disebut juga **Effect-free Programming,** di mana fungsi tidak mengubah variabel global, tidak melakukan operasi input/output, dan tidak mengubah data di luar ruang lingkupnya.

2. Pure Function

Untuk menciptakan sebuah kode yang menerapkan pendekatan Effect-free programming, kita dapat menggunakan teknik Pure Function. **Pure function** adalah sebuah fungsi yang selalu mengembalikan/me-return output yang sama ketika diberi input yang sama (sebagaimana contoh fungsi matematika sebelumnya). Artinya, untuk setiap inputan, fungsi ini akan **mengembalikan hasil yang sama secara konsisten,** tidak peduli kapan atau dimana fungsi dipanggil. Selain itu, pure function **tidak menyebabkan efek samping apa pun** selain me-return value. Artinya, pure function tidak memodifikasi state dalam sistem di luar scope-nya. Mirip-mirip dengan konsep Effect-free programming bukan? Yup,

hal ini dikarenakan Pure Function merupakan implementasi dari Effect-free programming. Perhatikan contoh berikut:

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

hasil =  tambah(3, 4)
print("Jumlah", hasil)
```

Dalam contoh di atas, fungsi ‘tambah’ merupakan **pure function.** Fungsi ‘tambah’ selalu menghasilkan output yang sama untuk input yang sama. Jika kita memanggil 'tambah(3, 4)', hasilnya akan selalu '7', tidak peduli berapa kali atau kapan fungsi ini dipanggil.

Selain itu, fungsi 'tambah' tidak memodifikasi state dalam sistem diluar scope. Fungsi ini hanya menerima dua argumen, yaitu a dan b, menjumlahkannya, dan mengembalikan hasilnya. Fungsi ini **tidak mengubah variabel global**, **tidak melakukan input-output**, atau memodifikasi data di luar ruang lingkupnya.

Dalam contoh lain, kita memiliki fungsi 'total_nilai' yang akan menghitung total nilai dari data yang sudah diberikan. Fungsi ini hanya bergantung pada parameter yang dibutuhkan, yaitu 'nilai'. Perhatikan contoh berikut:

```
nilai = [
    {'matkul': 'Fungsional', 'nilai': 95},
    {'matkul': 'Mobile', 'nilai': 55}
]

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)
```

Selanjutnya, kita memiliki fungsi 'tambah_nilai' yang digunakan untuk menambahkan nilai matkul lain ke dalam kumpulan nilai yang telah kita punya. Fungsi ini akan memodifikasi variabel nilai yang ada di luar 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!")

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

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

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

Fungsi 'tambah_nilai' **bukan** merupakan fungsi yang murni (pure function) karena fungsi ini memodifikasi variabel nilai yang ada di luar fungsi. Selain itu, jika kita panggil fungsi 'tambah_nilai' beberapa kali, isi dari variabel nilai akan mengalami perubahan incremental (tidak tetap). Sebagai pembuktian, coba jalankan code yang sama sekali lagi! Hasilnya akan berbeda dengan output code sebelumnya (perhatikan pada bagian nilai matkul Mobile). Itulah mengapa fungsi ini bukan termasuk pure function.

Bagaimana cara agar fungsi ‘tambah_nilai’ menjadi pure function? Untuk membuat pure function, jangan lupa untuk menghindari penggunaan variabel global atau penggunaan data di luar fungsi. Sebagai gantinya:

1. Fungsi harus menerima ‘nilai’ sebagai parameter
2. Fungsi harus mengembalikan objek ‘nilai’ yang telah dimodifikasi tanpa merubah data luar
3. Caranya adalah dengan membuat salinan (copy) objek ‘nilai’ untuk dimodifikasi kemudian dikembalikan menjadi output fungsi

Untuk penjelasan lebih lanjut mengenai pure functions, kalian bisa lihat video ini.

3. Pemrograman Deklaratif

Dalam paradigma pemrograman deklaratif, kita akan ditekankan pada kalimat: **"Pemrograman deklaratif berfokus pada apa (what) dibanding bagaimana (how)".** Maksudnya adalah, dalam pemrograman deklaratif, kita mendeklarasikan tujuan atau apa (what) hasil yang ingin kita capai, dibandingkan memikirkan bagaimana (how) langkah spesifik untuk mencapainya. Pemrograman fungsional adalah 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* melakukan sesuatu dengan langkah-langkah detail proses yang jelas. Perhatikan potongan kode berikut:

```
numbers = [1, 2, 3, 4, 5]
double_numbers = []
for number in numbers:
    double_numbers.append(number * 2)
print(double_numbers)
```

Output: [2, 4, 6, 8, 10]

Dalam program di atas, kita melihat langkah-langkah detail tentang bagaimana mencapai hasil, mulai dari menyiapkan variabel kosong `double_numbers` untuk menyimpan hasil, mendefinisikan loop yang mengiterasi list `numbers`, dan menambahkan hasil dari setiap elemen nya dikali dua ke dalam list `double_numbers`. 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* yang harus dicapai tanpa perlu tahu bagaimana cara mencapainya. Contoh:

```
# Mempersiapkan fungsi deklaratif

'''
desc: fungsi double(x) untuk merubah input x menjadi 2x
pre-cond: input x merupakan sembarang data, bisa int, float, dsb
post-cond: 2x sebagai return value
'''
def double(x):
  # Banyak cara untuk mendapatkan 2x. Terserah kita mau pakai rumus apa.
  # Hal ini tidak perlu dijelaskan dalam deskripsi fungsi.
  return x + x
```

Sebelum menjalankan kode program berikut, pastikan kalian telah menjalankan (*run*) program imperatif sebelumnya. Jika tidak, akan terjadi *error: name 'numbers' is not defined*. Karena data `numbers` berada disana.

```
# Memanfaatkan fungsi deklaratif untuk membangun program

double_numbers = map(double, numbers)
print(list(double_numbers))
```

Output: [2, 4, 6, 8, 10]

Logika yang sama, bahasa yang sama, dan output yang sama pula, namun kali ini lebih ringkas. Dalam program di atas, kita cukup 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* yang dilakukan oleh fungsi *map* dan *double*, tanpa perlu tau *bagaimana* cara mereka melakukannya. Lebih detail tentang fungsi *map* akan kita bahas pada modul selanjutnya.

Demikian juga dengan fungsi *double* yang 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 Iterator

Suatu proses yang diulang lebih dari sekali dengan menerapkan logika yang sama disebut dengan **Iterasi.** Dalam bahasa pemrograman Python, sebuah loop dibuat dengan beberapa kondisi untuk melakukan iterasi hingga memenuhi batas. Sedangkan **Iterator** itu sendiri merupakan sebuah objek yang berisi sejumlah nilai yang dapat dihitung dan digunakan untuk mengiterasi objek yang dapat diubah seperti list, tuple, set, dan lain-lain.

Iterator diimplementasikan menggunakan class dan variabel lokal untuk iterasi. Iterator mengikuti konsep ***lazy evaluation,*** dimana evaluasi dari ekspresi akan ditahan dan disimpan dalam memori hingga item tersebut dipanggil secara spesifik, yang membantu kita untuk menghindari evaluasi berulang. Dalam penggunaannya, iterator memiliki:

- 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.
- Setelah objek iterable selesai diiterasi menggunakan iterator, untuk menggunakannya kembali, objek tersebut perlu ditugaskan kembali (reassign) ke iterator baru. Ini karena **iterator hanya bisa digunakan sekali**; setelah selesai, ia tidak bisa di-reset atau digunakan ulang secara langsung.

```python
class EvenNumbers:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= self.limit:
            result = self.current
            self.current += 2
            return result
        else:
            raise StopIteration

# Membuat objek iterator
even_iter = EvenNumbers(10)

# Menggunakan iterator untuk mengiterasi dan mencetak
bilangan genap
for num in even_iter:
    print(num)
```

Sebagian besar objek di Python seperti list, tuple, string, dan lain-lain adalah iterable. Namun tidak semua yang iterable merupakan iterator, meski bisa diubah menjadi iterator. Bingung ya? lebih detailnya akan kita bahas selanjutnya dalam topik lazy evaluation.

Suatu objek dikatakan iterable jika dari objek tersebut bisa kita buat iterator. Caranya sama seperti melakukan casting, yakni dengan memanggil fungsi iter(). Contohnya sebagai berikut:

```python
# Mendefinisikan list
list_genap = [2, 4, 6, 8, 10]
print(list_genap)
print(type(list_genap))

# Membuat iterator dengan iter()
iterator_genap = iter(list_genap)
print(iterator_genap)
print(type(iterator_genap))
```

Output:

```
[2, 4, 6, 8, 10]
<class 'list'>
<list_iterator object at 0x7f109ae95ae0>
<class 'list_iterator'>
```

Iterator menggunakan konsep OOP (Object Oriented Programming) karena sesuai dengan definisi yang sudah disebutkan di awal: **iterator merupakan sebuah objek yang berisi sejumlah nilai yang dapat dihitung dan digunakan untuk mengiterasi objek yang dapat diubah.** Hal ini tentunya sangat bertentangan dengan konsep Pemrograman Fungsional yang berfokus pada penggunaan fungsi (function) sebagai komponen utama dalam menulis program. Oleh karena itu, kita menggunakan pendekatan lain dalam Python yang lebih berorientasi pada fungsi dan lebih sesuai dengan paradigma Pemrograman Fungsional, yaitu **Generator.**

4.2 Generator

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 memakai 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-tama kita perlu membuat sebuah fungsi generator dengan menambahkan keyword **'yield'** sebagai pengganti keyword *return.* Jangan lupa untuk menambahkan deklarasi/deskripsi pada setiap fungsi yang kita bangun!

```
"""
desc: fungsi generator 'bilangan_genap(limit)' akan menghasilkan bilangan
genap dari 0 hingga 'limit'.
pre-cond: 'limit' harus berupa int >= 0
post-cond: fungsi 'bilangan_genap' menghasilkan object generator yang
berisikan bilangan genap dari 0 sampai 'limit'
"""

def bilangan_genap(limit):
    n = 0
    while n <= limit:
        yield n
        n += 2
```

Pada kode di atas, fungsi ‘bilangan_genap(limit)’ adalah fungsi generator yang menghasilkan object generator yang berisikan bilangan genap dari 0 sampai 'limit'. Keyword ‘yield’ digunakan untuk mengembalikan nilai secara bertahap setiap kali data dari object generator dipanggil tanpa menghentikan eksekusi fungsi sepenuhnya.

Kemudian kita siapkan sebuah variabel ‘generator_genap’ sebagai keranjang penyimpanan hasil dari fungsi ‘bilangan_genap’:

*) Pastikan kalian sudah **run** kode fungsi ‘bilangan_genap’ sebelum **run** kode berikut! Agar tidak eror.

```
generator_genap = bilangan_genap(10)

print(type(bilangan_genap))
print(type(generator_genap))
print(bilangan_genap)
print(generator_genap)
```

Output:

```
<class 'function'>
<class 'generator'>
<function bilangan_genap at 0x7bad78e6b9a0>
<generator object bilangan_genap at 0x7bad78e91310>
```

'generator_genap' adalah objek generator yang dihasilkan dengan memanggil fungsi 'bilangan_genap(10)'. Kalian bisa membedakan keduanya dengan cara memanggil fungsi 'print' dan 'type'.

Object 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 (next(generator_genap))
print (next(generator_genap))

#looping akan melanjutkan proses generate dari angka genap terakhir
for num in generator_genap:
    print(num,end = ' ')

#eror disini karena objek generator telah habis/mencapai limit
print (next(generator_genap))
```

Output:

```
0
2
4 6 8 10
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-9-81c39ee70639> in <cell line: 7>()
      5     print(num,end = ' ')
      6
----> 7 print (next(generator_genap))  #eror disini karena objek generator telah
habis/mencapai limit

StopIteration:
```

Di sini kita menggunakan fungsi 'next(generator_genap)' untuk mengambil nilai berikutnya dari generator secara berurutan. Loop 'for num in generator_genap' juga dapat digunakan untuk mencetak setiap bilangan genap selanjutnya yang dihasilkan oleh generator.

Setelah program dieksekusi, maka akan dihasilkan angka genap kurang dari limit (10), yaitu: 0, 2, 4, 6, dan 8. 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 pun tidak bisa digunakan lagi.

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 dikenal dengan sebutan **Generator Expression.** Cara membuatnya cukup dengan menggunakan tanda kurung dan perulangan didalamnya (bisa juga ditambahkan kondisi if)**.** Perhatikan contoh berikut:

```
generator_genap = (i for i in range (21) if i % 2 == 0)
print(generator_genap)
```

Output:

```
<generator object <genexpr> at 0x7a29b675d000>
```

Kode di atas adalah cara yang sama untuk membuat iterasi dengan range 21 yang akan mencetak angka genap dari 0 sampai 20. Lebih sederhana lagi daripada harus membuat fungsi generator yang panjang seperti sebelumnya ataupun class iterator yang rumit bukan... Inilah kekuatan dari fungsional programming yang deklaratif.

- ➔ Kita cukup mendeklarasikan **apa** yang kita butuhkan tanpa perlu tahu detail **bagaimana** program akan mengeksekusinya. Bukankah dasar dari semua perangkat komputer pintar jaman sekarang seperti itu??!
- ➔ Dengan ini kita dapat meringkas penulisan kode program yang berbaris-baris menjadi lebih efektif dan efisien. Meski kadang menjadi sedikit rumit untuk dipahami, tapi bukankah yang sederhana memang tidak selalu mudah!!?

5. Lazy Evaluation

Dari tadi istilah ini sudah berulang kali disebutkan. Tapi apa sih sebenarnya maksudnya? Daripada banyak teori bertele-tele, sebaiknya kita langsung praktek saja biar lebih paham tentang apa itu lazy evaluation. Pertama-tama, mari kita lihat kode di bawah:

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

Kode di atas merupakan contoh penggunaan range dan list dalam python. Sekilas memang keduanya mirip, tetapi ternyata terdapat perbedaan antara keduanya. Pada range, kita menggunakan konsep **lazy evaluation,** di mana evaluasi ekspresi akan menghasilkan satu item pada satu waktu dan menghasilkan item hanya pada saat dibutuhkan. Item dari objek yang lazy tidak bisa diakses secara langsung. Seringkali kita butuh iterasi/loop untuk mengaksesnya satu persatu. Silahkan kalian coba print keduanya untuk membuktikan!

Dengan demikian kita bisa menganggap bahwa penggunaan range yang lazy pasti akan lebih hemat memori daripada list, mengingat data yang tidak bisa diakses berarti data tersebut belum tersedia di memory internal kita. Mari kita buktikan dengan menjalankan potongan kode berikut:

```
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
```

Namun, apakah kemudian objek range yang lazy bisa lebih hemat waktu eksekusinya juga dibanding dengan list? Mari kita buktikan:

```
import timeit
print(timeit.timeit('''r = range(1, 100)''', number = 1000000))
print(timeit.timeit('''lst = list(range(1, 100))''', number = 1000000))
```

Output:

```
0.2785158169999704
1.4991627220006194
```

Ternyata, waktu eksekusi range lebih singkat daripada list. Dapat disimpulkan bahwa range **lebih hemat memori serta hemat waktu daripada list.** Hal ini membuktikan bahwa sesuatu yang malas tidak selalu berkonotasi negatif. Khususnya pada pemrograman fungsional. Yang disebut lazy disini malah bisa bekerja lebih cepat dan lebih hemat. Konsep lazy evaluation merupakan salah satu teknik unggulan dalam paradigma fungsional. Keren kan fungsional...

CODELAB

CODELAB 1

a. Ubahlah fungsi 'tambah_nilai' yang ada pada halaman 4 bab Pure Function menjadi sebuah pure function sesuai dengan step yang sudah dijelaskan!
b. Tambahkan fungsi generator 'cetak_nilai' untuk menampilkan data nilai. Cetak data generator dengan looping for untuk nim GANJIL dan fungsi next() untuk nim GENAP!

CODELAB 2

Telah disediakan struktur data global berupa list 'data mahasiswa' yang digunakan untuk menghitung nilai mahasiswa. Setiap nilai direpresentasikan sebagai dictionary dengan atribut 'nama', 'matkul', dan 'nilai'.

```
data_mahasiswa = [
    {'nama': 'Karina', 'matkul': 'Pemrograman Fungsional', 'nilai': 90},
    {'nama': 'Seulgi', 'matkul': 'Pemrograman Mobile', 'nilai': 56},
    {'nama': 'Wonyoung', 'matkul': 'Pemrograman Web', 'nilai': 95},
    {'nama': 'Hanni', 'matkul': 'Piranti Cerdas', 'nilai': 88},
    {'nama': 'Jihyo', 'matkul': 'Jaringan Komputer', 'nilai': 63},
]
```

a. Buatlah fungsi ‘rata_rata’ yang menghasilkan nilai rata-rata dari kelima mahasiswa tersebut. Gunakan pure function dan implementasikan fungsi deklaratif.
b. Buatlah sebuah fungsi 'kelulusan' yang menerima input data_mahasiswa dan mengembalikan daftar baru dari data_mahasiswa dengan mengubah value nilai menjadi ‘sempurna’ untuk nilai 85 keatas, ‘memenuhi’ untuk nilai kurang dari 85 dan ‘gagal’ untuk nilai dibawah 60.
c. Buatlah sebuah fungsi generator untuk mencetak nilai dari daftar nilai mahasiswa tersebut sesuai NIM akhir kalian. Jika NIM akhir genap, maka cetaklah nilai yang genap saja. Jika NIM ganjil, maka cetaklah nilai ganjil saja.

CODELAB 3

Buat generator expression yang akan menyimpan bilangan kelipatan dari tanggal lahir kalian (menggunakan input) dalam range(1000) dan tampilkan 10 data pertama dengan ketentuan:

a. NIM genap: gunakan teknik casting list dan slicing
b. NIM ganjil: gunakan looping dan method next()

TUGAS PRAKTIKUM

TUGAS 1

Melanjutkan tugas modul 1 agar memenuhi paradigma fungsional.

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

Membuat sebuah program yang dapat memproses dan menganalisis data. Program **harus** mengimplementasikan **pure** function dan **deklaratif**!

*)Note: Adanya penggunaan/penambahan exception handling dalam program merupakan nilai tambah. Kalian dapat berkreasi pada bagian ini guna menghindari program identik.

NIM Genap:

Tentang data penjualan barang dengan ketentuan sebagai berikut:

1. Buatlah sebuah daftar atau list ‘data_penjualan’ yang berisi informasi seperti ID produk, nama produk, harga, jumlah terjual (quantity), dan tanggal penjualan yang disusun dalam struktur data dictionary. Berikut contoh list untuk satu tanggal **(tidak boleh identik dengan contoh di bawah) :**

```
data_penjualan = [
    {"id_produk": "GNxxx", "nama_produk": "Daster", "harga":
45000, "quantity": 2, "tanggal": "2024-08-05"},
    {"id_produk": "GNxxx", "nama_produk": "Blouse", "harga":
70000, "quantity": 5, "tanggal": "2024-08-05"},
    {"id_produk": "GNxxx", "nama_produk": "Celana", "harga":
75000, "quantity": 3, "tanggal": "2024-08-05"},
]
```

2. Kemudian buatlah fungsi ‘hitung_pendapatan’ yang mengembalikan daftar baru dari semua informasi produk yang disertai total pendapatan untuk setiap produk.

CATATAN

a. Harus terdapat 4 tanggal berbeda, di mana di setiap tanggal terdapat 3 produk berbeda yang terjual (total 12 produk).
b. ID produk diawali dengan GN diikuti 3 angka acak. Contoh GN123, GN456, dan seterusnya.

CONTOH OUTPUT:

```
Product ID: GNxxx
Nama Produk: Daster
Harga: 45000
Jumlah: 2
Tanggal: 2024-08-05
Pendapatan: 90000
```

*) di atas hanya contoh dari 1 data, sementara kalian harus memiliki 12 data. Format output bisa dikreasikan secara bebas.

3. Buatlah fungsi 'average_penjualan' yang akan mengembalikan rata-rata penjualan di tanggal tertentu sesuai inputan user. Jangan lupa **buatlah exception handling** apabila tanggal tidak ditemukan atau tidak sesuai format.

   Contoh output:

```
Masukkan tanggal yang ingin dicari (YYYY-MM-DD): 2024-08-08
Rata-rata penjualan pada tanggal 2024-08-08: Rp66818.18
```

```
Masukkan tanggal yang ingin dicari (YYYY-MM-DD): 0909-09-09
Tidak ada penjualan pada tanggal tersebut
```

4. Buatlah fungsi 'total_penjualan' sebagai generator dengan inputan data hasil 'hitung_pendapatan' yang akan menghasilkan penjualan per tanggal pada setiap produk. Contoh output:

```
Tanggal: 2024-08-05, Total Penjualan: 665000
Tanggal: 2024-08-06, Total Penjualan: 555000
Tanggal: 2024-08-07, Total Penjualan: 295000
Tanggal: 2024-08-08, Total Penjualan: 735000
```

NIM Ganjil:

Tentang data reservasi hotel dengan ketentuan sebagai berikut:

1. Buatlah sebuah daftar atau list 'data_penginapan' yang berisi room ID, customer name, expenses, jumlah orang, dan tanggal menginap yang disusun dalam struktur data dictionary! Berikut contoh list untuk satu tanggal **(data yang digunakan tidak boleh identik dengan contoh):**

```
data_penginapan = [
    {"room_id": "GJxxx", "cust_name": "Bocil Telat", "expenses": 150000,
"jumlah_orang": 3, "tanggal": "2024-08-03"},
    {"room_id": "GJxxx", "cust_name": "Amonali", "expenses": 120000,
"jumlah_orang": 2, "tanggal": "2024-08-03"},
    {"room_id": "GJxxx", "cust_name": "Windut", "expenses": 175000,
"jumlah_orang": 2, "tanggal": "2024-08-03"},
    {"room_id": "GJxxx", "cust_name": "Detsit", "expenses": 140000,
"jumlah_orang": 4, "tanggal": "2024-08-03"}
]
```

2. Kemudian buatlah fungsi ‘cari_customer’ sesuai inputan user yang akan mengembalikan semua informasi customer beserta total tagihan customer.

CATATAN

a. Harus terdapat 3 tanggal berbeda, di mana di setiap tanggal terdapat 4 customer berbeda (total 12 customer).
b. ID produk diawali dengan GJ diikuti 3 angka acak. Contoh GJ123, GJ456, dan seterusnya.
c. Terdapat exception handling apabila nama customer tidak ditemukan.

CONTOH OUTPUT:

```
Masukkan nama customer: Windut
ID Kamar: GJxxx
Nama: Windut
Tagihan: 175000
Jumlah Orang: 2
Tanggal: 2024-08-03
Total Tagihan: 350000
--------------------
```

```
Masukkan nama customer: Brando
Customer dengan nama 'Brando' tidak ditemukan
```

*) di atas hanya contoh dari 1 data, sementara kalian harus memiliki 12 data customer.

**) Format output bisa dikreasikan secara bebas.

3. Buatlah fungsi ‘rata_rata_menginap’ sebagai generator untuk menghitung rata-rata jumlah orang yang menginap tiap harinya.

   Contoh output:

```
Tanggal: 2024-08-03 , Rata-rata yang menginap: 2.75
```

4. Buatlah fungsi 'total_pendapatan' yang akan mengembalikan sebuah daftar baru yang berisikan tanggal, jumlah customer, dan pemasukan.

   Contoh output:

```
Tanggal: 2024-08-03, Jumlah Customer: 4, Total Pendapatan: 1600000
Tanggal: 2024-08-04, Jumlah Customer: 4, Total Pendapatan: 1715000
Tanggal: 2024-08-05, Jumlah Customer: 4, Total Pendapatan: 1660000
```

KRITERIA & DETAIL PENILAIAN

KETERANGAN		POIN	PROGRAM IDENTIK
Codelab	Codelab 1, 2, & 3 masing-masing 5 poin	15	15
Tugas 1	Pemahaman	25	20
	Code/Kelengkapan Fitur (pure & deklaratif)	15	0
Tugas 2	Pemahaman	25	22
	Code/Kelengkapan Fitur (pure & deklaratif)	20	0
Total		**100**	**57**

*)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).