VERSION 2.1

27 September 2024

[PEMROGRAMAN FUNGSIONAL]

Modul 3 – Processing Sequence Data in Functional Programming

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

TUJUAN

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

TARGET MODUL

1. Praktikan mampu mengelola data berdasarkan proses yang tepat sebagai bentuk komputasi data dengan memanfaatkan teknik-teknik fungsional (map, filter, reduce, dll)
2. Praktikan diharap dapat memanfaatkan list comprehension guna membangun struktur data yang lebih efektif dan efisien

PERSIAPAN SOFTWARE/APLIKASI

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

1. Teknik Pengolahan Data pada Paradigma Fungsional

Dalam paradigma fungsional, program kita merupakan sebuah urutan transformasi data dari suatu bentuk ke bentuk yang lain. Mari kita lihat beberapa contoh implementasi nyata tentang cara kerja paradigma fungsional ini dalam dunia kita (informatika):

1. Untuk pemrograman back-end server, kita dapat memodelkannya sebagai proses transformasi sebuah HTTP request menjadi HTTP response.
2. Untuk pemrograman front-end, kita dapat memodelkannya sebagai proses transformasi data dari server menjadi representasi UI.
3. Compiler merupakan program yang mentransformasi kode sumber menjadi executable.

Konsep transformasi data ini biasanya didukung dengan konsep data immutability/kekekalan data. Data yang kekal berarti bahwa kita tidak akan bisa mengubah nilainya setelah dibuat. Semua fungsi yang bekerja pada sebuah struktur data akan mengembalikan nilai baru, bukan memutasi nilainya. Ini merupakan poin penting dalam konsep transformasi data, karena kita akan bisa yakin bahwa data akan selalu kekal dan tidak mungkin diubah di tempat lain kecuali melalui fungsi transformasi kita. Konsep ini telah kita pelajari sebelumnya pada modul 2 tentang pure function sebagai syarat pemrograman fungsional. Itulah karakteristik ketiga dari paradigma fungsional: transformasi data, gunakan immutability, jangan mutasi data. Hal-hal inilah yang akan menjadi bahasan kita pada modul 3 kali ini.

Selain itu, saat kita bekerja dengan koleksi data (seperti range, list, tuple, dictionary, dll), terdapat dua pola pemrograman yang sangat umum muncul, antara lain:

1. Melakukan **iterasi koleksi untuk membangun koleksi lain**. Pada setiap iterasi, terapkan beberapa transformasi atau beberapa tes ke item saat ini dan tambahkan hasilnya ke koleksi baru. Ini adalah konsep dari map() dan filter()
2. Melakukan **iterasi dan proses akumulasi hasil untuk membangun nilai tungga**l. len(), min(), max(), sum(), dan reduce() adalah contoh dari konsep ini.

2. Lebih Lanjut tentang Data Sequence

Pemrograman fungsional sangat cocok untuk bekerja dengan data sequence, seperti list, tuple, dan range, yang mana telah kita pelajari di modul 1. Sebagaimana juga telah kita pelajari sebelumnya tentang iterator, kita dapat menggunakan iterable object seperti: range, list, tuple, dictionary, generator, dll sebagai parameter input fungsi iter() dan menjadikannya lazy. Masih ingat apa keuntungan dari kita menggunakan lazy evaluation? Coba cek kembali bahasan modul 2 sebelumnya!

Pada topik kali ini kita tidak akan mengulang bahasan yang sama yang telah dibahas di modul sebelumnya. Pun kita tidak akan membahas jenis tipe data sequence baru. Namun kita akan mencoba salah satu cara yang lebih efektif dalam membangun sebuah struktur data sequence, khususnya tipe list.

2.1. List Comprehension

Di modul-modul sebelumnya, kita mempelajari tentang list dan cara mendeklarasikannya. Seperti ini:

```
list_ganjil = [
1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49]
print(list_ganjil)
```

Untuk membuat list dengan ukuran yang lebih pendek, cara diatas memang cocok digunakan. Namun, apabila kita ingin membuat dengan isi yang lebih panjang, tentu cara ini tidak efisien. Misalnya jika kita akan membuat suatu list yang berisi angka ganjil mulai dari 1 sampai 100 atau bahkan 1000. Alih-alih menggunakan kode seperti di atas, kalian mungkin akan terpikir untuk menggunakan perulangan dan append list pada setiap iterasinya seperti berikut:

```
list_ganjil = []
for i in range(50):
  if i % 2 != 0:
    list_ganjil.append(i)
    print(i, end=',')
```

Output:

```
1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41,
43, 45, 47, 49,
```

Dengan demikian tujuan kita bisa tercapai dengan cara yang lebih efektif. Sepanjang apapun data yang kita inginkan dapat kita bangun dengan cara ini (cukup mengubah nilai input dari range). Namun cara ini tergolong imperatif dan tidaklah fungsional. Oleh karena itu, penggunaan list comprehension bisa menjadi opsi yang lebih baik daripada kedua kode di atas.

List comprehension adalah **cara ringkas untuk membuat** daftar (**list**) di Python dengan metode yang lebih ringkas dan mudah dibaca. Ini memungkinkan kita untuk membuat daftar baru dengan menerapkan ekspresi pada setiap item dalam iterable, serta menambahkan pilihan untuk memfilter item berdasarkan kondisi tertentu. Berikut format penggunaannya:

Syntax:

```
newlist = [ekspresi for item in iterable if kondisi == True]
```

Ekspresi disini merupakan value yang akan dimasukkan dalam list (append). Selebihnya (bagian iterasi-for dan kondisi-if) adalah kode yang sama persis yang kita susun secara linier (sebaris) menjadi seperti ini:

```
list_ganjil = [x for x in range (50) if x % 2 != 0]
print(list_ganjil)
```

Output:

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]
```

Lebih ringkas dan sederhana bukan. Cukup dengan satu baris kode sederhana kita dapat membuat daftar sepanjang yang kita mau. Kode diatas akan mengembalikan list baru tanpa mengubah iterable data. Hal ini sejalan dengan prinsip immutability pada pemrograman fungsional. Selain itu, konsep ini memenuhi karakteristik pemrograman fungsional sebagaimana yang telah dibahas di modul 2 sebelumnya, yaitu gaya deklaratif. Dengan list comprehension, kita menyatakan apa yang kita inginkan (daftar baru) daripada menjelaskan bagaimana cara mendapatkannya (dengan loop).

Secara keseluruhan, list comprehension adalah alat yang kuat dalam Python. Guna mewujudkan prinsip-prinsip pemrograman fungsional dengan

mempromosikan cara yang lebih ekspresif dan ringkas untuk memanipulasi list, menekankan immutabilitas dan abstraksi tingkat tinggi dibandingkan dengan teknik pemrograman imperatif tradisional.

2.2. Nested List

Nested List adalah list yang di dalamnya berisi list lain sebagai elemen. Dalam Python, nested list sering digunakan untuk merepresentasikan struktur data multidimensi, seperti matriks, tabel, atau grafik. Sebagaimana kita sadari bahwa data list pada python hampir mirip seperti konsep array pada bahasa pemrograman lain, maka nested list juga mirip seperti array multidimensi. Tentunya dengan beberapa kelebihannya (list dibanding array).

a. Percobaan 1: Membuat Matrix

Dalam contoh berikut, kita akan membuat matrix 4×5 menggunakan nested loop. Outer loop mengiterasi sejumlah baris matrix (yaitu empat kali), melakukan proses append sublist kosong ke dalam matrix. Sedangkan inner loop kita gunakan untuk mengisi setiap sublist dengan nilai mulai dari 0 hingga 4 (sebanyak 5 elemen), sehingga menghasilkan matrix dengan nilai integer yang berurutan.

Tanpa List Comprehension

```
matrix1 = []

for i in range(4):
  matrix1.append([])
  for j in range(5):
    matrix1[i].append(j)
print(matrix1)
```

Output:

```
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

CHALLENGE

Sekarang coba kalian tampilkan nested list di atas agar lebih tampak seperti matrix 4 baris 5 kolom! Setelah berhasil, tunjukkanlah pada asisten!

Menggunakan List Comprehension

Sekali lagi, penggunaan looping adalah tidak disarankan dalam paradigma fungsional. Karena pemrograman fungsional punya teknik/cara yang lebih sederhana dan deklaratif daripada itu. Ya, hasil yang sama juga bisa didapatkan menggunakan nested list comprehension. Cukup hanya satu baris kode seperti ini:

```
matrix1 = [[j for j in range(5)] for i in range(4)]
print(matrix1)
```

Output:

```
[[0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4], [0, 1, 2, 3, 4]]
```

Coba perhatikan struktur syntax untuk nested list comprehension berikut:

Syntax:

newlist = [[*ekspresi* for *item* in *inner_iterable*] for *item* in *outer_iterable*]

Disini kita memasukkan sebuah list comprehension ([*ekspresi* for *item* in *inner_iterable*]) sebagai ***ekspresi*** dari list comprehension lain. Dan untuk statement kondisi-if adalah opsional, bisa ditambahkan, bisa juga tidak.

CHALLENGE

Buatlah sebuah list comprehension untuk menyimpan data seperti berikut:

```
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

dan tampilkan dalam bentuk matrix!

b. Percobaan 2: Filtering Nested List Menggunakan List Comprehension

Penambahan kondisi-if pada bagian akhir dari list comprehension memungkinkan kita menyaring data dalam nested list sesuai kriteria yang diinginkan. Hal ini dikenal juga dengan istilah filtering. Contohnya, kita dapat mengambil hanya elemen dengan nilai genap atau ganjil, atau melakukan operasi filtering lainnya berdasarkan baris atau kolom dari nested list.

Dalam sintaks nested list comprehension, terdapat dua list comprehension. Masing-masing dapat dilengkapi dengan kondisi if yang berbeda, dan akan memberikan dampak yang berbeda pula. Dimana kondisi-if pada inner list akan mempengaruhi bagian kolom matrix, dan kondisi-if pada outer list akan berpengaruh pada pemilihan/filter baris.

Secara konvensional, pengkondisian menggunakan if dapat dituliskan didalam/setelah loop yang kita inginkan seperti berikut:

```
# Didefinisikan baris dan kolom sesuai ukuran matrix
baris = 3
kolom = 4
```

Tanpa List Comprehension

```
# Membuat nested list 'matrix2_genap' hanya dengan elemen genap
matrix2_genap = []
for y in range(baris):
    row = []
    for x in range(y * kolom, (y + 1) * kolom):
        if x % 2 == 0:
            row.append(x)
    matrix2_genap.append(row)
print("Data list genap:\n\t",matrix2_genap)

# Membuat nested list 'matrix2_ganjil' yang berisi elemen ganjil pada baris genap
matrix2_ganjil = []
for y in range(baris):
    if y % 2 == 0:  # Kondisi untuk memilih hanya baris genap
        row = []
        for x in range(y * kolom, (y + 1) * kolom):
            if x % 2 == 1:  # Kondisi untuk memilih elemen ganjil
                row.append(x)
        matrix2_ganjil.append(row)
print("Data list ganjil pada baris genap:\n\t",matrix2_ganjil)
```

Menggunakan List Comprehension

```
# Print out hasil chalenge
print("Data list asli (tanpa filter):\n\t", matrix2)

# Menambahkan filter genap untuk semua elemen matrix
matrix2_genap = [[x for x in range(y*kolom, (y+1)*kolom) if x%2==0]
                 for y in range(baris)]
print("Data list genap:\n\t", matrix2_genap)

# Membuat filter elemen ganjil pada baris genap
matrix2_ganjil = [[x for x in range(y*kolom, (y+1)*kolom) if x%2==1]
                  for y in range(baris) if y%2==0]
print("Data list ganjil pada baris genap:\n\t", matrix2_ganjil)
```

Kedua kode diatas akan menghasilkan data output yang sama seperti ini:

```
Data list asli (tanpa filter):
        [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
Data list genap:
        [[0, 2], [4, 6], [8, 10]]
Data list ganjil pada baris genap:
        [[1, 3], [9, 11]]
```

c. Percobaan 3: Mengakses Elemen dalam Nested List

Seperti yang telah disebutkan sebelumnya, Nested list tidak jauh berbeda dengan array multidimensi. Secara struktur bahkan mereka sama persis. Bisa kalian perhatikan pada kode berikut! Bahkan cara mengakses elemen nya pun sama, menggunakan index.

```python
percobaan3 = [
    [2, 5, 7, 9, 3],
    [0, 1, 9, 3, 5],
    [7, 1, 4, 6, 7],
    [0, 9, 7, 3, 2]
]

# Mengakses elemen baris kedua, kolom ketiga
elemen1 = percobaan3[1][2]
print("Baris kedua kolom ketiga: ", elemen1)

# Mengakses elemen baris ketiga, kolom kedua
elemen2 = percobaan3[2][1]
print("Baris ketiga kolom kedua: ", elemen2)
```

Output:

```
Baris kedua kolom ketiga:  9
Baris ke tiga kolom ke dua:  1
```

Pada konteks pemrograman fungsional, kita dapat memanipulasi nested list menggunakan konsep-konsep seperti pure function, list comprehension, dan fungsi-fungsi built-in seperti map, filter, dan reduce yang akan menjadi pembahasan kita juga pada modul 3 ini.

3. Teknik Mengolah Data Sequence

Saat kita bekerja dengan koleksi data/data sequence pasti akan melibatkan iterasi/perulangan. Terdapat dua pola pemrograman yang sangat umum muncul terkait pengolahan data sequence, yaitu (1) **iterasi untuk membuat koleksi baru** dan (2) **iterasi plus komputasi untuk mendapatkan nilai tunggal** dari data. Sebagaimana yang telah kita

pelajari sebelumnya (modul 2) bahwa pemrograman fungsional merupakan paradigma pemrograman deklaratif yang berfokus pada '**apa**' daripada **'bagaimana'**. Sehingga untuk proses iterasi data sequence pada pemrograman fungsional, kita tidak lagi menggunakan looping (for/while) dan menuliskan langkah-langkah imperatif program. Namun disini kita akan menggunakan teknik atau fungsi-fungsi deklaratif untuk mencapai apa yang kita inginkan.

Beberapa teknik pengolahan data sequence yang umum digunakan antara lain adalah fungsi map(), filter(), dan reduce(). Fungsi-fungsi ini memerlukan sebuah **fungsi dan iterator sebagai argumen**/parameter input. Dalam paradigma pemrograman fungsional, fungsi yang bisa **menerima fungsi lain sebagai input** dikenal dengan istilah fungsi orde tinggi (*Higher-Order Function*). Fungsi map(), filter(), dan reduce() merupakan *built-in Higher-Order Function* bawaan dari Python.

Higher-Order Function, immutability dan lazy evaluation merupakan teknik andalan paradigma fungsional dan cara yang efisien untuk memproses data sequence. Dari ketiga keyword tersebut, yang mungkin masih asing bagi kalian adalah Higher-Order Function (HoF). Penjelasan detail tentang apa itu HoF, akan kita bahas pada modul selanjutnya. Sedangkan pada modul ini kita akan berfokus pada fungsi map(), filter(), dan reduce() sebagai built-in HoF yang bisa kita gunakan secara langsung di python.

3.1. Filter

Sebelumnya kita sudah mengimplementasikan filtering pada list comprehension dengan menambahkan kondisi-if di dalamnya. Dengan list comprehension saja sebenarnya sudah cukup bisa untuk kita melakukan filtering data. Namun, di modul 2 kemarin kita sudah mengenal satu konsep andalan paradigma fungsional, yaitu lazy evaluation. Dan data list tidak lazy. Jika ada yang bisa dikerjakan secara lazy (dengan bermalas-malasan namun lebih optimal), kenapa tidak!? Itulah tujuan dari fungsi filter() disini.

Fungsi ini dinamai sebagaimana peruntukannya, yaitu fungsi yang digunakan untuk **menyaring elemen-elemen dari sebuah iterable data** (seperti list, tuple, atau string) berdasarkan suatu kondisi atau kriteria tertentu. Fungsi 'filter' menerima dua argumen, yaitu **sebuah fungsi dan iterable data/object** sebagai parameter input**.**

Fungsi yang bisa dijadikan sebagai input filter() harus berupa **fungsi logika** yang berisi suatu kondisi tertentu untuk menyaring data. Fungsi ini akan diterapkan ke setiap elemen dalam data iterable. Untuk setiap elemen yang memenuhi kondisi 'True' akan diloloskan dan disertakan dalam hasil akhirnya (sebagai return value). Oleh karena itu, umumnya **hasil dari filter akan lebih sedikit dari data semula.** Namanya juga disaring (filter) kan.

Mari kita coba mempraktekkan proses filtering data secara fungsional berikut:

```
#1. Menyiapkan fungsi logika
'''
desc: fungsi_genap(x) menerima sebuah bilangan dan mengembalikan nilai boolean True
jika bilangan tersebut adalah genap, dan False jika bilangan tersebut adalah ganjil.

pre-cond: input x merupakan sembarang data, bisa int, float, dlsb
post-cond: fungsi akan mengembalikan True jika x adalah bilangan genap,
           dan False jika x adalah bilangan ganjil.
'''
def fungsi_genap(x):
    return x % 2 == 0

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

#3. Pengunaan fungsi filter()
angka_genap = filter(fungsi_genap, numbers)
print(angka_genap)
```

Output:

```
<filter object at 0x78ff5b737a60>
```

Seperti yang sudah disebutkan sebelumnya, fungsi filter adalah implementasi dari lazy evaluation. Oleh karena itu, hasil dari filter tidak bisa di-print secara langsung. Seperti halnya generator yang lazy. Untuk mencetak nilai hasil filter, kita bisa memperlakukannya layaknya generator atau iterator., yakni dengan menggunakan fungsi next() maupun dengan iterasi atau loop seperti berikut:

```
# Mengakses filter object menggunakan next
print('First item:',next(angka_genap))
print('Second item:',next(angka_genap))
```

```
# Mengakses map object menggunakan iterasi
print('The rest item:')
for item in angka_genap:
  print(item, end = ' ')
```

Output:

```
First item: 2
Second item: 4
The rest item:
6 8 10 12 14
```

Atau bisa juga dilakukan dengan parsing data ke tipe lain seperti list atau tuple. Karena data ‘numbers’ kita hanyalah deret angka berurut dari 1 sampai 15, maka kita dapat dengan mudah menggantikannya dengan data range yang lazy juga daripada list ‘numbers’. Masih ingat materi modul 2 tentang ini kan. Saat kita membandingkan antara data range dan data list. Silahkan perhatikan contoh di bawah ini:

```
# Membuat object filter baru menggunakan range sebagai iterable data
new_numbers = filter(fungsi_genap, range(1, 15))

# Casting filter object menjadi list untuk menampilkan hasil
print(list(new_numbers))
```

Output:

```
[2, 4, 6, 8, 10, 12, 14]
```

Karena data numbers kita hanyalah deret angka berurut dari 1 sampai 15, kita dapat dengan mudah menggantikannya dengan data range yang lebih lazy daripada list. Masih ingat materi modul 2 tentang ini kan. Saat kita membandingkan antara data range dan data list.

Hasil yang sama persis bisa kita peroleh dengan baris kode yang lebih singkat dan bahkan lebih hemat memory karena semuanya lazy evaluation. Super sekali paradigma fungsional ini!

Eits, tunggu sampai kalian bertemu lambda di modul 4. Dengan itu nantinya kita tidak perlu lagi membuat deklarasi fungsi. Sabar. Jangan terburu-buru, jangan ya dek ya, pelan-pelan dulu saja kita. Yang penting paham, okay? Lanjut.

3.2. Map

Sesuai dengan namanya, map adalah peta. Map digunakan untuk **memetakan data**. Fungsi map() menerapkan sebuah **fungsi transformasi** ke setiap item dalam sebuah iterable data dan mengembalikan sebuah objek map. Hasil dari map ini selalu sama dalam konteks jumlah/panjang data dengan iterable object yang diberikan.

Karena fungsinya untuk memetakan, maka kita **hanya boleh memasukkan fungsi yang akan merubah data** dan bukan fungsi logika**,** melainkan **fungsi transformasi.** Contoh penggunaannya seperti di bawah ini:

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

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

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

Output:

```
<map object at 0x78ff5b736860>
```

Sama seperti generator dan filter(), fungsi map juga mengimplementasikan **lazy evaluation.** Oleh karena itu, hasil dari map tidak bisa di-print secara langsung. Untuk mencetak nilai, perlu diperlakukan pula seperti iterator dan generator objek seperti di bawah:

```
# Mengakses map object menggunakan next
print('First item:', next(doubled_numbers))
print('Second item:', next(doubled_numbers))

# Mengakses map object menggunakan iterasi
print('The rest item:')
for item in doubled_numbers:
  print(item, end = ' ')
```

Output:

```
First item: 2
Second item: 4
The rest item:
6 8 10
```

Karena sama, maka disini kita juga bisa melakukan parsing objek map ke dalam tipe sequence lain. Seperti ini contohnya:

```
# Membuat object map baru karena double_numbers sudah tidak bisa digunakan lagi
# karena item telah habis digenerate pada iterasi sebelumnya
new_numbers = map(double, range(1,6))

# Casting map object menjadi tuple untuk menampilkan hasil
print(tuple(new_numbers))
```

Output:

```
(2, 4, 6, 8, 10)
```

Bagaimana dengan data non-number seperti misal String. Apakah juga bisa kita implementasikan fungsi map untuk data String? Tentu saja bisa, selama dua syarat argumen yang dibutuhkan oleh fungsi map terpenuhi. Yaitu:

1. Ada fungsi untuk transformasi data.
2. Data yang diberikan merupakan data sequence.

Bukankah String juga sequence of char? Berarti bisa langsung kita jadikan data input untuk dimappingkan kan. Mari coba buktikan! Jalankan dan amati kode berikut:

```
# Map single string menggunakan fungsi double
map_str = map(double,'hai')
print(list(map_str))
```

Bagaimana dengan list of string:

```
# Map list of string menggunakan fungsi double
name_list = ['andi','budi','dewi']
double_name = map(double,name_list)
print(tuple(double_name))
```

Sebagaimana deskripsi dari fungsi double sebelumnya:

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

Fungsi ini menerima x sebagai sembarang data, maka data tipe string pun bisa kita kirimkan sebagai input fungsi. Dan kemudian ditransformasikan oleh fungsi double menjadi 2x. Jadilah seperti itu hasilnya. Bagaimana jika kita mencoba dengan fungsi lain yang lebih String friendly?

```
'''
desc: fungsi uppercase(s) merubah string input (s) menjadi huruf besar
pre-cond: input s harus bertipe string
post-cond: string input akan dikembalikan dalam bentuk huruf besar
'''
def uppercase(s):
  return s.upper()

caps_name = map(uppercase,name_list)
print(list(caps_name))
```

Output:

```
['ANDI', 'BUDI', 'DEWI']
```

Apa yang terjadi jika fungsi uppercase() kita gunakan untuk memetakan data numbers? Pasti akan terjadi error, karena data input tidak sesuai dengan precondition fungsi. Silahkan jalankan kode berikut untuk mengetahui hasilnya:

```
caps_name = map(uppercase,numbers)
print(list(caps_name))
```

Jika fungsi map bisa untuk data String, pastinya filter() pun juga bisa. Cukup siapkan fungsi yang relevan dengan data sebagai input nya.

Menggunakan Map untuk Operasi pada Nested List

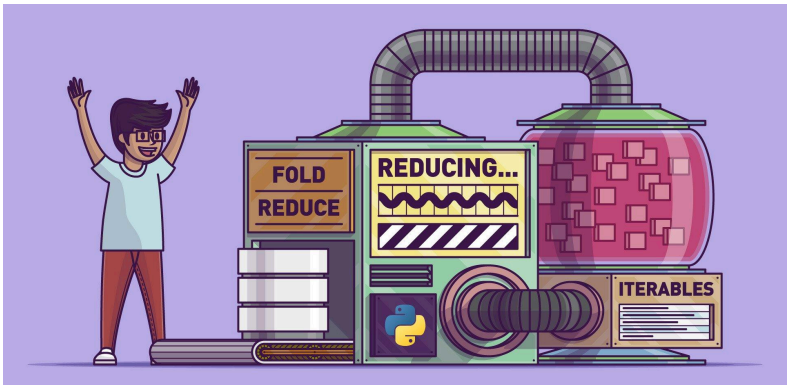
Setelah mempelajari konsep dasar fungsi map() yang memungkinkan kita menerapkan sebuah fungsi pada setiap elemen dalam suatu list secara efisien, kita dapat melangkah lebih jauh dengan menerapkannya pada nested list. Fungsi map() juga dapat digunakan untuk menerapkan sebuah fungsi ke setiap elemen dari nested list secara fungsional. Perhatikan contoh berikut:

```
# Disini kita akan menggunakan fungsi double dan data list matrix
# dari source code sebelumnya

print("List Asli: ",matrix)

doubleMatrix = [list(map(double, list_baris)) for list_baris in matrix]
print("Hasil Map :", doubleMatrix)
```

Output:

```
List Asli:  [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
Hasil Map : [[0, 2, 4, 6], [8, 10, 12, 14], [16, 18, 20, 22]]
```

3.3. Reduce



Fungsi reduce() pada Python adalah fungsi yang mengimplementasikan teknik matematika yang disebut reduksi atau *folding* (pelipatan). Kita akan mereduksi atau melipat kumpulan data menjadi satu nilai sebagai output dari reduce(). Sedikit berbeda dengan dua fungsi sebelumnya (map dan filter), reduce() bisa menerima tiga argumen. Dua argumen pertama mirip seperti pada map dan filter, yaitu **sebuah fungsi dan iterable object/data.** Sedangkan argumen ketiga dari reduce adalah opsional. Kita bisa tambahkan ataupun tidak perlu dituliskan dalam penggunaan reduce. Ini akan kita bahas berikutnya.

Reduce berguna saat kita perlu menerapkan sebuah fungsi ke suatu iterable data dan mereduksinya menjadi satu nilai output. Dengan demikian, maka fungsi yang akan menjadi input argumen reduce harus yang menerima dua argumen untuk dioperasikan (secara aritmatik maupun komparatif) dan mengembalikan satu nilai.

Argumen yang diperlukan untuk membuat reduce() adalah **fungsi dua argumen** dan **iterable.** Fungsi ini akan diaplikasikan ke item yang ada pada iterable untuk dihitung secara aritmatik maupun komparatif menjadi suatu nilai akhir. Argumen kedua yang diperlukan adalah iterable, fungsi reduce dapat menerima semua iterable Python. Termasuk list, tuple, range, object, generator, iterator, set, key dan value dictionary, and segala object Python yang dapat diiterasi.

Untuk memahami bagaimana reduce() bekerja, coba perhatikan kode berikut. Pertama, kita harus menyiapkan sebuah fungsi:

```
'''
desc: Fungsi untuk menghitung jumlah dari a + b
pre-cond: a dan b bisa berupa sembarang data
post-cond: print out dan hasil dari a+b sebagai return value
'''
def pertambahan(a, b):
  hasil = a + b
  print(f"{a} + {b} = {hasil}")
  return hasil

pertambahan(10, 7)
```

Output:

```
10 + 7 = 17
17
```

pertambahan() adalah fungsi dua argumen, yang akan mengakumulasi dua input menjadi satu hasil. Sehingga fungsi ini cocok untuk dijadikan sebagai argumen dari fungsi reduce. Selanjutnya kita cukup menambahkan sebuah iterable sebagai argumen kedua reduce() untuk dihitung jumlah akumulasi dari semua itemnya.

Sebelum menggunakan fungsi reduce, kita perlu menambahkan sebuah module/library tempat reduce berada. Nama module nya adalah functools. Modul Functools diperuntukkan bagi fungsi tingkat tinggi (Higher Order Function) yang bekerja pada fungsi lain. Modul ini menyediakan fungsi untuk bekerja dengan fungsi lain. Salah satunya adalah reduce. Perhatikan cara import modul dan penggunaan reduce pada contoh berikut:

```
from functools import reduce

angka = [1, 3, 5, 7]
hasil = reduce(pertambahan, angka)
print(hasil)
```

Output:

```
1 + 3 = 4
4 + 5 = 9
9 + 7 = 16
16
```

Saat kita memanggil reduce() dengan melempar fungsi pertambahan dan 'angka' sebagai argumen, kita akan mendapat output yang menunjukkan semua operasi yang dijalankan reduce() hingga menghasilkan nilai akhir 16. Dalam kasus ini, operasi ini ekuivalen dengan (((1 + 3) + 5) + 7) = 16.

Pemanggilan reduce() dalam contoh di atas berlaku pertambahan() untuk dua item pertama dalam 'angka' (1 dan 3) kemudian menghasilkan 4 sebagai hasil output. Kemudian reduce() memanggil pertambahan() menggunakan output sebelumnya (4) dan item berikutnya dalam 'angka' (yaitu 5) sebagai argumen, menghasilkan 9. Proses ini akan terus berulang hingga jumlah item habis dan reduce() menghasilkan output hasil perhitungan terakhir, yaitu 16.

Selain dua argumen (fungsi dan iterable) tadi, seperti yang sudah disebutkan sebelumnya, reduce bisa menerima hingga tiga argumen. **Argumen ketiga dalam reduce() adalah initializer.** Argumen ini bersifat **opsional**. Jika kita memberikan sebuah nilai pada initializer ini, maka reduce() akan memasukkannya ke pemanggilan fungsi pertama sebagai argumen pertamanya. Artinya, pemanggilan pertama ke fungsi akan menggunakan nilai initializer dan item pertama dari iterable untuk melakukan komputasi parsial pertamanya. Setelah itu, reduce() terus bekerja dengan item iterable berikutnya.

Berikut contoh penggunaan reduce dengan initializer diset ke 100, jalankan dan amati hasilnya:

```
reduce(pertambahan, angka, 100)
```

Menggunakan Reduce untuk Nested List

Setelah mempelajari bagaimana fungsi reduce() digunakan untuk melakukan agregasi nilai dari sebuah list menjadi satu hasil akhir, apakah kita juga dapat menerapkannya dalam konteks nested list? Mari kita coba dengan menjalankan kode berikut dan amati hasilnya!

```
print(reduce(pertambahan,matrix2))
```

Output:

```
[0, 1, 2, 3] + [4, 5, 6, 7] = [0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7] + [8, 9, 10, 11] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Reduce tidak dapat mengakses inner elemen dalam nested iterable/nested list. Reduce hanya akan mengoperasikan fungsi input (pertambahan) kedalam apapun isi dari iterable. Dalam kasus ini, isi dari iterable nested list adalah sebuah list. Maka reduce akan melakukan proses pertambahan dari inner list pertama ([1, 2, 3, 4]) dengan inner list kedua ([5, 6, 7, 8]). Proses pertambahan (+) dalam list sama halnya dengan concatenation pada string. Sehingga menghasilkan gabungan dari dua list menjadi satu (menjadi [1, 2, 3, 4, 5, 6, 7, 8]) dan seterusnya.

Bagaimana jika kita ingin menjumlahkan seluruh angka pada ‘matrix’? Maka kita harus menjumlahkan dulu seluruh elemen dalam inner list satu persatu, baru kemudian menjumlahkan kembali hasil penjumlahan dari masing-masing inner list tadi. Sehingga disini kita perlu melakukan dua kali proses reduce. Mulai bingung? Lihat contoh di bawah!

```
# Buat fungsi baru tanpa print agar lebih sederhana
'''
desc: Fungsi untuk menghitung jumlah dari x + y
pre-cond: x dan y bisa berupa sembarang data
post-cond: hasil dari x+y sebagai return value
'''
def tambah(x, y):
  return x+y

print(matrix2) # Tampilkan isi matrix terlebih dahulu
# Implementasikan reduce pada setiap baris matrix
jumlah_elemen = [reduce(tambah, baris) for baris in matrix2]
print("First reduce: ", jumlah_elemen)
```

```
# Reduce yang kedua untuk menjumlahkan seluruh jumlah_elemen
hasil_akhir = reduce(tambah, jumlah_elemen)
print("Second reduce :", hasil_akhir)
```

Output:

```
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
First reduce:  [6, 22, 38]
Second reduce : 66
```

Secara konsep, kode diatas sangat bisa untuk kita sederhanakan lagi menjadi hanya satu baris kode. Jika memang tujuan kita hanya ingin tahu hasil penjumlahan dari seluruh elemen dalam 'matrix'. Tanpa perlu memperhatikan bagaimana proses dalam pencapaiannya. Sebagaimana konsep paradigma fungsional seperti itu bukan.. Adakah yang tahu bagaimana caranya? Mari kami tunjukkan satu baris kode untuk mencapai target kita tadi:

```
hasil = reduce(tambah, [reduce(tambah, baris) for baris in matrix2])
print(hasil)
```

Output:

```
66
```

4. Fungsi Rekursif

Perhitungan faktorial seringkali dijadikan contoh untuk membuat fungsi rekursif.

```
'''
desc: fungsi untuk menghitung nilai faktorial dari sebuah angka n
pre-cond: input n merupakan sebuah bilangan/number
post-cond: nilai faktorial dari n
'''
def faktorial_rekursif(n):
  if n <= 1:
    return 1  # Base case
  else:
    return n * faktorial_rekursif(n - 1)  # Recursive case

print(faktorial_rekursif(5))
```

Output:

```
120
```

Fungsi rekursif memiliki dua komponen utama:

1. Base Case: Kondisi yang menentukan kapan rekursi berhenti. Tanpa base case, fungsi rekursif akan memanggil dirinya sendiri tanpa henti, yang menyebabkan stack overflow (kondisi yang terjadi ketika tumpukan (stack) memori program terisi penuh).
2. Recursive Case: Bagian dari fungsi yang memanggil dirinya sendiri untuk menyelesaikan sub-masalah. Disinilah iterasi terjadi.

Rekursi hampir mirip dengan konsep reduce. Dimana reduce memanggil fungsi yang sama berulang kali untuk mencapai tujuan. Bedanya, rekursif memanggil dirinya sendiri secara berulang, dan bukan fungsi lain.

Sebenarnya kita bisa menggunakan list comprehension, map, filter, dan reduce untuk menghindari looping secara eksplisit dan menjaga kode agar tetap bersih dan deklaratif. Perhitungan faktorial di atas juga bisa diselesaikan menggunakan fungsi reduce. Apa iya? Ga percaya? Begini caranya:

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

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

Output:

```
120
```

Namun, ada **beberapa kasus** dimana kita tidak bisa menggunakan teknik-teknik tersebut. Ketika kita **perlu menerapkan logika kompleks atau kondisi yang lebih dari sekadar satu langkah,** kita dapat menggunakan fungsi rekursif untuk mencapai tujuan. Seperti pada logika kasus penyusunan baris angka fibonacci.

Fibonacci adalah salah satu contoh umum dari masalah yang dapat diselesaikan dengan rekursif. Setiap elemen dalam deret Fibonacci didefinisikan sebagai jumlah dari dua elemen sebelumnya. Pada dasarnya, Fibonacci merupakan fungsi matematika yang dihitung dengan cara rekursif. Sehingga otomatis penyelesaiannya pun seharusnya dengan cara mendefinisikan fungsi rekursif daripada iterasi biasa. Seperti ini caranya

untuk mengubah iterasi for menjadi rekursif dalam kasus Fibonacci, amati dan perhatikan dimana perbedaanya:

```
# Fungsi untuk membuat n baris fibonacci (tanpa rekursif)
# dengan n adalah angka bulat positif
def fibonacci_sequence_loop(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

    seq = [0, 1]
    for i in range(2, n):
        seq.append(seq[-1] + seq[-2])
    return seq

# Contoh penggunaan
print(fibonacci_sequence_loop(n))  # Menghasilkan 10 angka pertama
```

Output:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
# Fungsi untuk membuat n deret fibonaci dengan rekursif
# dengan n adalah angka bulat positif
def fibonacci_recursive(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
      seq = fibonacci_recursive(n-1)
      return seq + [seq[-1]+seq[-2]]

print(fibonacci_recursive(n))
```

Output:

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Meskipun looping dalam bentuk tradisional tidak diterapkan dalam pemrograman fungsional. Serta tidak semua persoalan bisa diselesaikan dengan beberapa teknik yang ada (seperti list comprehension, map, filter, dan reduce). Fungsi rekursi adalah satu teknik yang bisa kita gunakan dalam menyusun program secara fungsional. Hal ini

membantu menjaga konsistensi dengan prinsip-prinsip pemrograman fungsional dan meminimalkan perubahan state dalam program.

CODELAB

CODELAB 1

1. Buatlah sebuah fungsi dua parameter: width dan height. Fungsi ini harus mengembalikan sebuah **nested list** yang merepresentasikan sebuah papan catur dengan pola tertentu (**menggunakan list comprehension**) berukuran sesuai dengan width dan height.

   Contoh*:

   ```
   [['o', 'x', 'o', 'x', 'o'],
    ['x', 'o', 'x', 'o', 'x'],
    ['o', 'x', 'o', 'x', 'o'],
    ['x', 'o', 'x', 'o', 'x'],
    ['o', 'x', 'o', 'x', 'o']]
   ```

2. Manfaatkan fungsi map untuk mengubah pola dari papan catur pada soal nomor 1 menjadi pola lain yang berbeda!

   Contoh*:

   ```
   [['#', '+', '#', '+', '#'],
    ['+', '#', '+', '#', '+'],
    ['#', '+', '#', '+', '#'],
    ['+', '#', '+', '#', '+'],
    ['#', '+', '#', '+', '#']]
   ```

   *) Simbol/char yang digunakan boleh bebas sesuai kreativitas masing-masing!

CODELAB 2

Diketahui data buku sebagai berikut:

```
books = [
    {'judul':'Pulang', 'penulis': 'Tere Liye', 'halaman': 400},
    {'judul':'Kapan Nanti', 'penulis': 'Ziggy Z.', 'halaman': 142},
    {'judul':'Namaku Alam', 'penulis': 'Leila S. Chudori', 'halaman': 448},
    {'judul':'Origin', 'penulis': 'Dan Brown', 'halaman': 511},
    {'judul':'Rumah Lebah', 'penulis': 'Ruwi Meita', 'halaman': 288},
    {'judul':'Kubah', 'penulis': 'Ahmad Tohari', 'halaman': 184},
    {'judul':'Dompet Ayah Sepatu Ibu', 'penulis': 'J. S. Khairen', 'halaman': 210}]
```

1. Buatlah sebuah daftar buku baru yang memenuhi kriteria buku berawalan huruf 'K' **menggunakan list comprehension.**
2. Buat daftar baru yang berisi hanya judul buku saja menggunakan fungsi map()*.
3. Buat daftar buku dengan halaman lebih dari 200 menggunakan fungsi filter()*.
4. Hitunglah total jumlah halaman semua buku menggunakan fungsi reduce()*.

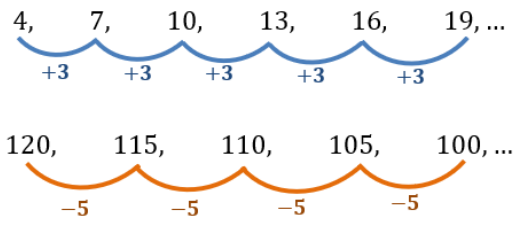
   *) **Dilarang menggunakan lambda.** Gunakan pure function serta implementasi fungsi deklaratif menjadi nilai tambah.

TUGAS PRAKTIKUM

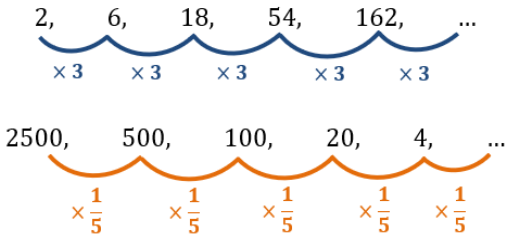
TUGAS 1

Barisan merupakan suatu urutan angka atau bilangan dari kiri ke kanan dengan pola serta aturan tertentu. Barisan berkaitan erat dengan deret. Jika barisan adalah kelompok angka atau bilangan yang berurutan, deret merupakan jumlah dari suku-suku pada barisan. Barisan dan deret terbagi menjadi beberapa macam. Namun, yang lebih umum diketahui adalah barisan dan deret Aritmetika dan Geometri.

Barisan aritmetika merupakan barisan bilangan yang memiliki beda atau selisih (*difference*-d) tetap antara dua suku yang berurutan. Sedangkan barisan geometri merupakan barisan bilangan dimana dua suku yang berurutan memiliki perbandingan yang sama. Perbandingan pada barisan geometri disebut sebagai rasio/*ratio* (r). Masing-masing memiliki rumus untuk menghitung suku ke-n (Un).



(a) contoh barisan aritmetika



(b) contoh barisan geometri

Pilih salah satu dari dua soal berikut untuk dikerjakan!

SOAL A (INTERMEDIATE)

1. Manfaatkan materi modul 3 ini untuk **membuat** dua buah **baris aritmetika dan geometri**. Kalian bisa menggunakan **list comprehension, map, filter, atau rekursif** sesuai kreativitas masing-masing.
2. Kemudian gunakan **reduce** untuk menghitung deret aritmetika dan geometri dari baris bilangan yang kalian buat pada soal nomor 1.

SOAL B (ADVANCED)

1. Buatlah sebuah **fungsi rekursif** untuk **membuat baris aritmetika-geometri** hingga suku ke-n. Dimana sangat mungkin untuk mengkombinasikan baris aritmatika dan baris geometri dalam satu urutan. Kombinasi ini sering disebut sebagai **baris aritmetika-geometri**. Disini, setiap suku dihasilkan dari kombinasi dari suku aritmatika dan suku geometri. Rumus suku ke-n dapat didefinisikan sebagai:

$$a_n = (a + (n-1) \times d) \times r^{(n-1)}$$

Dimana: a = Suku pertama,
d = Beda (aritmetika),
r = Rasio (geometri)

Contoh:

```
a = 2  # Suku pertama
d = 3  # Beda aritmetika
r = 2  # Rasio geometri
n = 5  # Menghitung suku hingga ke-5

result = arithmetic_geometric_sequence(a, d, r, n)
print(result)
```

```
[2, 10, 32, 88, 224]
```

2. Gunakan **reduce** untuk menghitung deret dari baris bilangan tersebut!

TUGAS 2 (LIVE CODING)

Disini tugas kalian adalah melanjutkan tugas modul 1 dan 2 sebelumnya dengan mengimplementasikan materi modul 3 pada project tugas kalian tersebut. Asisten akan memandu dan memberikan tugas terkait **list comprehension, map, filter, reduce dan rekursif** sesuai dengan data dan fungsi-fungsi yang kalian miliki dari tugas modul

sebelumnya untuk dapat kalian kerjakan secara live code. Pengerjaan tugas ini bergantung pada pemahaman dan kreativitas kalian. Yang tentunya **tidak boleh identik dengan praktikan lain!**

*) **Dilarang menggunakan lambda.** Gunakan pure function serta implementasi fungsi deklaratif menjadi nilai tambah.

KRITERIA & DETAIL PENILAIAN

KETERANGAN		POIN**	PROGRAM IDENTIK*/**
Codelab 1	masing-masing soal bernilai 2,5 poin	5	5
Codelab 2	masing-masing soal bernilai 2,5 poin	10	10
Tugas A Intermediate	Kreativitas kode	5	0
	Fungsionalitas & Pemahaman	10	10
Tugas B Advanced	Kreativitas kode	10	0
	Fungsionalitas & Pemahaman	25	20
Tugas 2	Kreativitas kode	20	10
	Fungsionalitas & Pemahaman	30	22
Total Codelab + Tugas A + Tugas 2		**80**	**57**
Total Codelab + Tugas B + Tugas 2		**100**	**67**

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

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