Modul_7_PBF

April 27, 2025

1 Modul 7 Praktikum Pemrograman Berbasis Fungsi: Modul Itertools

Tujuan Praktikum: - Mahasiswa dapat memahami dan menerapkan fungsi itertools tak berhingga - Mahasiswa dapat memahami dan menarapkan fungsi itertools berhingga - Mahasiswa dapat menyelesaikan analisis data dengan penggunaan modul itertools

1.1 Pendahuluan

Dalam pemrograman fungsional, kita lebih mengutamakan objek yang *stateless* (tidak memiliki keadaan yang berubah-ubah) dan menggunakan struktur data yang tidak bisa diubah (*immutable*). Di Python, pendekatan ini mendorong kita untuk menggunakan:

- **Generator expression**
- **Generator function**
- **Iterable**

Daripada menggunakan struktur data besar yang bersifat mutable (seperti list, set, dict), Python menyediakan **modul `itertools`** sebagai solusi efisien untuk mengelola data dalam bentuk iterable.

1.1.1 Tujuan Penggunaan `itertools`

Modul `itertools` menyediakan berbagai **fungsi iterator** yang membantu dalam: - Pemrosesan data berukuran besar - Menghindari penggunaan memori yang berlebihan - Penulisan kode yang lebih ekspresif dan ringkas

1.2 Konsep Dasar Iterables dan Iterators

Sebelum memahami `itertools`, penting untuk memahami dua konsep dasar:

1. **Iterable**

- Objek yang dapat di-*loop* (dilakukan iterasi) seperti `list`, `tuple`, `set`, `str`, atau generator.
- Dapat diubah menjadi `iterator` dengan fungsi `iter()`.

2. **Iterator**

- Objek yang mengikuti *Iterator Protocol*, yaitu memiliki metode `__next__()` dan `__iter__()`.
- Hanya bisa digunakan **satu kali** sampai habis (*exhausted*).
- Setelah habis, hanya akan melemparkan `StopIteration`.

Batasan Penting: Iterator Hanya Sekali Pakai

"Iterables can be used only once."

- **Iterator tidak memiliki panjang (`len()`).**
- **Tidak bisa digunakan ulang** — jika sudah habis, maka akan kosong dan tidak menghasilkan apa-apa lagi.
- Python tidak memberikan error eksplisit saat iterator yang sudah habis digunakan kembali.

1.3 Kelompok Fungsi dalam `itertools`

1.3.1 Fungsi untuk Iterator Tak Berhingga

Digunakan untuk *iterable* tanpa batas ukuran, cocok untuk stream data atau perhitungan tak terbatas.

Contoh fungsi: - `itertools.count(start=0, step=1)` → menghitung tanpa akhir - `itertools.cycle(iterable)` → mengulangi isi iterable tanpa henti - `itertools.repeat(object, times=None)` → mengulangi objek terus-menerus atau hingga `times`

1.3.2 Fungsi untuk Iterator Terbatas (Finite)

Digunakan untuk memanipulasi dan *mereduksi* data iterable yang berukuran terbatas.

Contoh: - `itertools.chain(*iterables)` → menggabungkan beberapa iterable - `itertools.compress(data, selectors)` → menyaring data berdasarkan `True/False` dari `selectors` - `itertools.groupby(iterable, key=None)` → mengelompokkan elemen berdasarkan `key`

1.3.3 Fungsi `tee()` untuk Menyalin Iterator

- `itertools.tee(iterable, n=2)` membuat `n` salinan independen dari satu iterator.
- Berguna saat iterator perlu digunakan lebih dari sekali.
- **Perhatian**: penggunaan `tee()` **memori intensif**. Lebih baik mengubah desain jika memungkinkan.

1.4 Fitur Tambahan Iterables dan Iterators

Fitur	Penjelasan
`len()`	Tidak dapat digunakan pada iterator
`next()`	Digunakan untuk mendapatkan elemen berikutnya
`iter()`	Membuat iterator dari iterable
`for` loop	Menyembunyikan detail teknis dengan menggunakan `iter()` secara otomatis

Modul `itertools` merupakan bagian penting dalam pemrograman fungsional Python karena:

- Efisien: memungkinkan pemrosesan data besar dengan memori minimal.
- Fleksibel: menyediakan banyak fungsi untuk kombinasi, pengulangan, penyaringan, dan pengelompokan data.
- Elegan: membuat kode lebih ringkas, ekspresif, dan Pythonic.

1.5 Fungsi Iterator Tak Berhingga dari `itertools`

Modul `itertools` menyediakan fungsi untuk membuat iterator tak berhingga. Kita akan mempelajari tiga fungsi utama:

1.5.1 `count(start=0, step=1)`

Iterator ini seperti `range()` tetapi tanpa batas atas.

Contoh:

```
[ ]: from itertools import count
     from itertools import islice

     # Menampilkan 5 angka pertama dari count
     list(islice(count(10, 2), 5))
     # Output: [10, 12, 14, 16, 18]
```

Bisa digunakan untuk mengganti `enumerate()`:

```
[ ]: enumerate = lambda x, start=0: zip(count(start), x)
     list(enumerate(iter('word')))
     # Output: [(0, 'w'), (1, 'o'), (2, 'r'), (3, 'd')]
```

Gunakan `count(start, step)` untuk menghitung dengan langkah tertentu:

```
[ ]: list(zip(count(1, 3), iter('word')))
     # Output: [(1, 'w'), (4, 'o'), (7, 'r'), (10, 'd')]
```

1.5.2 `cycle(iterable)`

Mengulangi iterable selamanya.

Contoh:

```
[ ]: from itertools import cycle
     from itertools import islice

     list(islice(cycle('abc'), 7))
     # Output: ['a', 'b', 'c', 'a', 'b', 'c', 'a']
```

1.5.3 `repeat(elem, [n])`

Mengulangi satu elemen berkali-kali, bisa tanpa batas.

Contoh:

```
[ ]: from itertools import repeat

     list(repeat(10, 3))
     # Output: [10, 10, 10]
```

1.6 Penanganan `count()` dengan Float

`count()` bisa digunakan dengan nilai float: **Akurasi**: float tidak selalu tepat secara biner, sehingga error bisa terakumulasi.

```
[ ]: from itertools import count
     list(islice(count(0.5, 0.1), 5))
     # Output: [0.5, 0.6, 0.7, 0.8, 0.9]
```

Solusi: Gunakan integer dan dikalikan:

```
[ ]: (x * 0.1 for x in count())
```

Fungsi `find_first()`: Cari elemen pertama dari iterator

```
[ ]: from collections.abc import Callable, Iterator
     from typing import TypeVar

     T = TypeVar("T")

     def find_first(
         terminate: Callable[[T], bool],  # Fungsi kondisi, menerima T dan␣
      ↪mengembalikan bool
         iterator: Iterator[T]             # Iterator dari elemen bertipe T
     ) -> T:                               # Fungsi ini mengembalikan elemen bertipe T
         i = next(iterator)               # Ambil elemen berikutnya dari iterator
         if terminate(i):                # Jika memenuhi kondisi (terminate)
             return i                    # Kembalikan elemen tersebut
         return find_first(terminate, iterator)  # Jika tidak, cari terus secara␣
      ↪rekursif
```

1.6.1 Studi Kasus: Akurasi Float

Membandingkan: - `count(0, 0.1)` → penjumlahan float langsung - `x * 0.1 for x in count()` → hitung integer dulu, baru dikali

```
from itertools import count
from typing import NamedTuple

Pair = NamedTuple('Pair', [('flt_count', float), ('int_count', float)])

source = (
    Pair(fc, ic) for fc, ic in
    zip(count(0, 0.1), (0.1 * c for c in count()))
)

def not_equal(pair: Pair) -> bool:
    return abs(pair.flt_count - pair.int_count) > 1.0E-12
```

```
find_first(not_equal, source)
# Output: Pair(flt_count=92.799..., int_count=92.800...)
```

Perbedaan kecil muncul karena representasi biner 0.1 tidak presisi.

1.6.2 Eksperimen Akurasi

Mendeteksi perbedaan float pertama:

```
source = map(Pair, count(0, 0.1), (0.1 * c for c in count()))
find_first(lambda pair: pair.flt_count != pair.int_count, source)
# Output: Pair(flt_count=0.6, int_count=0.6000000000000001)
```

Hal ini menunjukkan bahwa perbedaan terkecil bisa muncul setelah beberapa iterasi, sekitar 1e-16 (2 ^{3}), batas presisi float 64-bit IEEE.

Catatan Penting - Gunakan integer untuk iterasi dan kalikan dengan float **untuk menjaga akurasi**. - Python **tidak mengoptimalkan rekursi tail-call**, sehingga `find_first()` bisa terbatas pada 1000-an iterasi.

1.6.3 Project Euler Problem #1

Soal: > Temukan jumlah semua bilangan positif di bawah 1000 yang habis dibagi 3 **atau** 5.

Solusi dengan `cycle()`

Sekarang kita akan **coba sesuatu yang lebih kreatif dan menyenangkan**: gunakan `itertools.cycle()` untuk menandai kelipatan 3 dan 5.

[ ]:
```
from itertools import cycle

def euler1_with_cycle(limit=1000):
    # Siklus FizzBuzz akan menandai bilangan mana yang habis dibagi 3 atau 5
    fizz = cycle([0, 0, 1])  # 1 setiap 3 kali
    buzz = cycle([0, 0, 0, 0, 1])  # 1 setiap 5 kali

    total = 0
    for i in range(1, limit):
        if next(fizz) or next(buzz):
            total += i
    return total

print(euler1_with_cycle())  # Output: 233168
```

Penjelasan: - `cycle([0, 0, 1])` artinya: urutan `[0, 0, 1, 0, 0, 1, ...]` untuk menandai kelipatan 3. - `cycle([0, 0, 0, 0, 1])` artinya: urutan `[0, 0, 0, 0, 1, 0, 0, 0, 0, 1, ...]` untuk kelipatan 5. - Jika salah satu `next(fizz)` atau `next(buzz)` bernilai 1, berarti `i` habis dibagi 3 atau 5.

1.7 Fungsi Iterator Berhingga dari Itertools

Python menyediakan **`itertools` module**, yang berisi banyak fungsi berguna untuk **memproses data dalam bentuk iterables**, seperti list, tuple, atau generator. Modul ini sangat membantu dalam manipulasi data, terutama saat bekerja dengan sequence yang besar, kompleks, atau hasil parsing file seperti CSV, XML, HTML, dan lainnya. Fungsi-fungsi itertools (dan built-in) menghasilkan output dengan jumlah elemen terbatas (finite). Artinya, mereka tidak terus-menerus menghasilkan nilai (berbeda dengan infinite iterators seperti `count()` atau `cycle()`).

- Mengenal **fungsi-fungsi penting dalam `itertools` dan built-in** yang memudahkan kita melakukan:
  - Pengurutan & pengelompokan
  - Penomoran elemen
  - Penggabungan dan penguraian iterables
  - Filtering data
  - Mapping ke fungsi

Tabel berikut memuat fungsi itertools berhingga

Kategori	Fungsi	Deskripsi Singkat
Built-in itertools	`enumerate()`	Memberi nomor urut pada item dalam iterable
	`accumulate()`	Mengembalikan hasil akumulasi (jumlah, produk, dll)
	`chain()`	Menggabungkan beberapa iterable secara berurutan
	`groupby()`	Mengelompokkan data berdasarkan kunci tertentu
	`zip_longest()`	Menggabungkan iterable seperti `zip()`, tapi pad jika panjang tidak sama
	`compress()`	Memfilter iterable berdasarkan iterable Boolean lain
	`islice()`	Memotong iterable seperti operasi slice
	`dropwhile()` & `takewhile()`	Memfilter item berdasarkan kondisi awal
	`filterfalse()`	Kebalikan dari `filter()`, ambil item yang gagal tes
	`starmap()`	Menerapkan fungsi pada tuple yang di-unpack seperti `f(*args)`

1.7.1 Menetapkan Angka dengan `enumerate()`

`enumerate()` adalah **fungsi built-in** di Python yang **menambahkan nomor indeks ke item dalam iterable**, dan menghasilkan pasangan `(index, item)`.

Format umum:

```
enumerate(iterable, start=0)
```

- `iterable`: objek yang bisa diiterasi (list, tuple, string, dst.)
- `start`: angka awal untuk penomoran (default = 0)

Contoh dasar:

```
data = ['apel', 'jeruk', 'mangga']
for i, buah in enumerate(data):
    print(i, buah)
```

Output:

```
0 apel
1 jeruk
2 mangga
```

1.8 Studi Kasus: `enumerate()` + `sorted()`

```
raw_values = [1.2, .8, 1.2, 2.3, 11, 18]
tuple(enumerate(sorted(raw_values)))
```

Langkah-langkahnya: 1. `sorted(raw_values)` → `[0.8, 1.2, 1.2, 2.3, 11, 18]` 2. `enumerate(...)` → `(0, 0.8), (1, 1.2), ...` 3. `tuple(...)` → kita ubah menjadi tuple lengkap.

Output:

```
((0, 0.8), (1, 1.2), (2, 1.2), (3, 2.3), (4, 11), (5, 18))
```

Cara ini sangat berguna jika kita ingin tahu: - posisi setiap item dalam urutan baru (misalnya setelah sorting), - atau memberi "ranking" sementara pada data.

Contoh: Penomoran untuk Baris Data

Dalam data nyata (misalnya file CSV, XML, HTML), kita kadang ingin menambahkan **nomor baris** atau **ID** ke setiap entri agar bisa dilacak kembali. Contohnya:

```
for i, row in enumerate(file_lines, start=1):
    print(f"Baris {i}: {row}")
```

Contoh Lanjutan: Trip Data & Leg Numbering

```
from typing import NamedTuple

class Point(NamedTuple):
    latitude: float
    longitude: float

class Leg(NamedTuple):
    order: int
    start: Point
    end: Point
    distance: float
```

Dengan `enumerate()`, kita buat setiap *leg* punya `order` atau urutan:

```
def numbered_leg_iter(pair_iter):
    for order, (start, end) in enumerate(pair_iter):
        yield Leg(
```

```
            order,
            start,
            end,
            round(haversine(start, end), 4)
        )
```

Memberi **urutan eksplisit** pada setiap rute perjalanan, sangat berguna kalau data-nya nanti di-*sort* dan kita ingin tetap tahu urutan aslinya.

1.9 Total Akumulasi dengan `accumulate()`

`accumulate()` adalah fungsi dari modul `itertools` yang digunakan untuk **menghitung akumulasi dari elemen-elemen dalam iterable**—dalam bentuk *running total* atau hasil *penggabungan bertahap* dari elemen-elemen tersebut.

Sederhananya: `accumulate([1, 2, 3, 4])` → `[1, 3, 6, 10]`
1 → 1+2 → 1+2+3 → 1+2+3+4

Sintaks:

```
from itertools import accumulate

accumulate(iterable, func=operator.add)
```

- `iterable`: list, tuple, atau generator
- `func`: fungsi untuk mengakumulasi, default-nya adalah `operator.add` (penjumlahan)

Contoh Dasar:

```
from itertools import accumulate

data = [1, 2, 3, 4]
result = list(accumulate(data))
print(result)
```

Output:

```
[1, 3, 6, 10]
```

Alternatif fungsi:

```
import operator

data = [1, 2, 3, 4]
list(accumulate(data, operator.mul))  # Running product
```

Output:

```
[1, 2, 6, 24]
```

1.9.1 Studi Kasus: Quartiling Data Perjalanan

Kita kembali ke konteks sebelumnya, yaitu data perjalanan (trip) yang terdiri dari objek `Leg`, masing-masing punya `distance`.

Tujuan kita: **mengelompokkan waypoint ke dalam 4 kuartil berdasarkan jarak kumulatif perjalanan**.

Langkah demi langkah:

1. Ambil jarak per leg:

```
distances = (leg.distance for leg in trip)
```

2. Hitung total kumulatif jarak tiap leg:

```
from itertools import accumulate

distance_accum = list(accumulate(distances))
```

Misalnya:

```
[5, 8, 10, 15, 18, 24, 30]
```

3. Hitung *scaling factor* untuk kuartil:

```
import math

scale = math.ceil(distance_accum[-1] / 4)
```

- `distance_accum[-1]`: total seluruh jarak
- `math.ceil(... / 4)`: membagi ke dalam 4 kuartil
- Gunakan pembulatan ke atas agar tidak menghasilkan nilai 4 (tidak ada kuartil ke-5)

4. Hitung kuartil per leg:

```
quartiles = list(int(d / scale) for d in distance_accum)
```

Contoh hasil:

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

Berarti: - Leg ke-0 dan 1 → kuartil 0 - Leg ke-2 dan 3 → kuartil 1 - dst.

Kombinasikan dengan Data Asli

Kita bisa `zip()` kuartil dengan objek `Leg`:

```
for q, leg in zip(quartiles, trip):
    print(f"Quartile {q}: {leg}")
```

Atau gunakan `groupby()` untuk mengelompokkan:

```
from itertools import groupby

for q, group in groupby(zip(quartiles, trip), key=lambda x: x[0]):
    print(f"Quartile {q}")
    for _, leg in group:
        print("  ", leg)
```

1.10 Menggabungkan Iterator dengan `chain()`

Fungsi `chain()` dari modul `itertools` digunakan untuk **menggabungkan beberapa iterator menjadi satu** iterator besar, seolah-olah semua datanya berasal dari satu sumber saja. Sederhananya:

```
from itertools import chain

a = [1, 2, 3]
b = [4, 5]
c = [6]

list(chain(a, b, c))  # [1, 2, 3, 4, 5, 6]
```

1.10.1 Studi Kasus: Gabungkan Beberapa File CSV

Misalnya kita punya banyak file CSV **berformat tab-delimited (`\t`)**, dan ingin membaca semua baris dari file-file itu **seolah-olah berasal dari satu file besar.**

Jika kita membuka tiap file secara manual dengan `with`, kodenya akan sangat berulang.
Contoh yang tidak efisien:

```
with open('file1.tsv') as f1, open('file2.tsv') as f2:
    for row in csv.reader(f1, delimiter="\t"):
        ...
    for row in csv.reader(f2, delimiter="\t"):
        ...
```

Solusinya: `ExitStack` + `chain()` + `csv.reader`

Penjelasan Fungsi:

```
from itertools import chain
import csv
from contextlib import ExitStack
from pathlib import Path
from typing import Iterator, TextIO
```

Fungsi utama:

```
def row_iter_csv_tab(*filepaths: Path) -> Iterator[list[str]]:
    with ExitStack() as stack:
        files: list[TextIO] = [
            stack.enter_context(path.open())
            for path in filepaths
        ]

        readers = map(
            lambda f: csv.reader(f, delimiter='\t'),
            files
        )
```

```
        yield from chain(*readers)
```

Apa yang terjadi di atas?

Bagian	Penjelasan
`ExitStack()`	Membuat satu konteks `with` untuk beberapa file
`stack.enter_context(path.open())`	Membuka tiap file dan memasukkannya ke stack agar ditutup otomatis nanti
`map(lambda f: csv.reader(...))`	Membuat pembaca CSV untuk setiap file
`chain(*readers)`	Gabungkan semua pembaca CSV jadi satu aliran baris
`yield from ...`	Menghasilkan tiap baris dari file secara satu per satu

Kenapa tidak `return`?

Kalau kita menulis:

```
return chain(*readers)
```

Maka generator dikembalikan **setelah** blok `with` selesai, sehingga semua file langsung **tertutup**, dan kita akan dapat error saat mencoba membacanya.

Solusinya: **`yield from` di dalam blok `with`** agar file tetap terbuka selama baris-barisnya diproses.

Contoh Penggunaan:

```
paths = [Path("file1.tsv"), Path("file2.tsv"), Path("file3.tsv")]

for row in row_iter_csv_tab(*paths):
    print(row)
```

code di atas akan mencetak semua baris dari tiga file TSV, seolah-olah datanya berasal dari satu file.

Hal-hal yang harus diperhatikan:

- `chain()` sangat berguna untuk menggabungkan iterable apa pun.
- `ExitStack` memungkinkan kita membuka banyak file dan menutupnya secara otomatis di akhir blok `with`.
- Menggabungkan `ExitStack + chain()` sangat cocok untuk memproses banyak file sebagai satu sumber data kontinu.
- Harus hati-hati dengan konteks `with`; gunakan `yield` bukan `return`.

1.11 Mempartisi Iterator dengan `groupby()`

Fungsi `groupby()` dari modul `itertools` digunakan untuk **mengelompokkan item dalam iterator** berdasarkan **nilai kunci (key)**.

Tetapi penting:
Hanya item **yang bersebelahan dan memiliki key yang sama** yang akan dikelompokkan.
Jadi, **data harus diurutkan terlebih dahulu berdasarkan key-nya**.

Pola Umum:

```
from itertools import groupby

# Pastikan data sudah diurutkan berdasarkan kunci
for key, group in groupby(data, key=fungsi_kunci):
    print(f"Kelompok {key}:")
    for item in group:
        print(item)
```

1.11.1 Contoh: Mengelompokkan Data Berdasarkan Kuartil

```
from itertools import groupby
from Chapter07.ch07_ex1 import get_trip

source_url = "file:./Winter%202012-2013.kml"
trip = get_trip(source_url)
quartile = quartiles(trip)

# Gabungkan kuartil dengan data trip
group_iter = groupby(zip(quartile, trip), key=lambda q_raw: q_raw[0])

for group_key, group in group_iter:
    print(f"Grup {group_key + 1}: {len(list(group))} perjalanan")
```

Apa yang terjadi? - `zip(quartile, trip)` membuat pasangan `(kuartil, data)`. - `groupby(..., key=lambda q_raw: q_raw[0])` akan mengelompokkan berdasarkan **nilai kuartil**. - Setiap kelompok adalah iterator dari perjalanan yang termasuk dalam kuartil yang sama.

Catatan Penting: `groupby()` **tidak** akan bekerja dengan benar jika data **tidak diurutkan terlebih dahulu**.
Jika tidak diurutkan, hasil pengelompokan bisa **salah atau terpecah**.

Alternatif: `defaultdict` untuk Pengelompokan

Kalau kamu **tidak ingin mengurutkan data dulu**, kamu bisa pakai cara lain:

```
from collections import defaultdict
from typing import Iterator, Callable, Hashable, TypeVar, Iterable

DT = TypeVar("DT")
KT = TypeVar("KT", bound=Hashable)

def groupby_2(
    iterable: Iterable[DT],
    key: Callable[[DT], KT]
) -> Iterator[tuple[KT, Iterator[DT]]]:
    groups: dict[KT, list[DT]] = defaultdict(list)
    for item in iterable:
        groups[key(item)].append(item)
    for k, items in groups.items():
        yield k, iter(items)
```

Kelebihan `defaultdict`: - **Tidak perlu diurutkan**. - Bisa mengelompokkan semua item yang memiliki kunci sama, walaupun posisinya acak.

Kekurangan `defaultdict`: - Menggunakan lebih banyak **memori** karena semua data disimpan dalam list. - Bisa jadi lambat untuk dataset yang sangat besar.

Perbandingan Singkat

Fitur	`groupby()`	`defaultdict()`
Harus diurutkan?	Ya	Tidak
Penggunaan memori	Hemat (lazy / streaming)	Lebih boros (semua data disimpan)
Urutan penting?	Ya	Tidak
Cocok untuk	Data besar yang terurut	Data acak yang ingin digabung semua

1.12 Menggabungkan Iterable dengan `zip()` dan `zip_longest()`

`zip()` Fungsi `zip()` digunakan untuk **menggabungkan beberapa iterable** (seperti list atau tuple) menjadi satu iterable baru berupa tuple-tuple.

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
list(zip(a, b))
# Output: [(1, 'a'), (2, 'b'), (3, 'c')]
```

> `zip()` **berhenti di iterable yang paling pendek**.

`zip_longest()` Berbeda dengan `zip()`, fungsi `zip_longest()` dari modul `itertools` akan **terus menggabungkan sampai iterable terpanjang habis**, dan **mengisi kekosongan dengan nilai default (`None` atau `fillvalue`)**.

```
from itertools import zip_longest

a = [1, 2]
b = ['a', 'b', 'c']
list(zip_longest(a, b))
# Output: [(1, 'a'), (2, 'b'), (None, 'c')]
```

Bisa diberi nilai pengganti:

```
list(zip_longest(a, b, fillvalue='?'))
# Output: [(1, 'a'), (2, 'b'), ('?', 'c')]
```

Kapan Gunakan `zip()` vs `zip_longest()`?

Kasus Penggunaan	Gunakan
Iterable sama panjang	`zip()`
Panjang tidak diketahui / berbeda	`zip_longest()`
Perlu padding untuk data yang hilang	`zip_longest()`
Analisis statistik eksploratif (EDA)	Gunakan dengan hati-hati karena padding bisa mengganggu makna statistik

Catatan Penting > Untuk **analisis data**, penggunaan `zip_longest()` harus dilakukan **dengan hati-hati**, karena: - Data yang dipad (diisi) bisa bersifat **tidak valid secara statistik** - Padding seperti `None`, `0`, atau `''` bisa **mengubah hasil analisis atau perhitungan**

Contoh Lain: Grouper Recipe

Python Standard Library mendokumentasikan "grouper recipe" yang sering menggunakan `zip_longest()` untuk membagi data menjadi kelompok.

```
from itertools import zip_longest

def grouper(iterable, n, fillvalue=None):
    "Mengelompokkan iterable ke dalam tuple berukuran n"
    args = [iter(iterable)] * n
    return zip_longest(*args, fillvalue=fillvalue)

data = [1, 2, 3, 4, 5, 6, 7]
list(grouper(data, 3))
# Output: [(1, 2, 3), (4, 5, 6), (7, None, None)]
```

1.13 Menyaring Data dengan `compress()`

Fungsi `compress()` dari modul `itertools` digunakan untuk **memilih elemen dari sebuah iterable berdasarkan iterable Boolean paralel (selector)**. fungsi ini berbeda dengan fungsi built-in `filter()` yang menggunakan **fungsi predikat** (function) untuk menentukan elemen mana yang lolos. Dengan `compress()`, **setiap nilai `True` dalam selector berarti elemen pada posisi yang sama akan dipertahankan**, dan `False` berarti dibuang.

Contoh Sederhana

```
from itertools import compress

data = ['a', 'b', 'c', 'd']
selector = [True, False, True, False]

list(compress(data, selector))
```

Output:

```
['a', 'c']
```

Studi Kasus: Seleksi Acak & Subset

Berikut ini beberapa contoh menggunakan `compress()` untuk menyaring data berdasarkan aturan:

```
import random
from itertools import compress, cycle

random.seed(1)
data = [random.randint(1, 12) for _ in range(12)]

def all_rows():
    return cycle([True])
```

```
def subset(c):  # Ambil 1/c dari elemen
    return cycle([True] + [False] * (c - 1))

def randomized(c):  # Acak elemen yang akan diambil
    return (random.randint(1, c) == 1 for _ in range(len(data)))
```

Semua data:

```
list(compress(data, all_rows()))
```

Ambil 1 dari setiap 3 data (subset):

```
list(compress(data, subset(3)))
```

Ambil elemen secara acak (1 dari setiap 3):

```
list(compress(data, randomized(3)))
```

1.13.1 Implementasi Konsep `filter()` dengan `compress()`

Fungsi `filter()` pada dasarnya bisa dibangun kembali menggunakan `compress()` dan `map()`, seperti ini:

```
from itertools import compress, tee
from typing import Callable, Iterable, Iterator, TypeVar

SrcT = TypeVar("SrcT")

def filter_concept(function: Callable[[SrcT], bool],
                   source: Iterable[SrcT]) -> Iterator[SrcT]:
    i1, i2 = tee(source)
    return compress(i1, map(function, i2))
```

Fungsi `tee()` digunakan untuk menggandakan iterable sehingga `i1` digunakan untuk output dan `i2` digunakan untuk evaluasi fungsi filter.

Keunggulan `compress()`

Fitur	Keterangan
Seleksi berbasis pola	Dapat membuat aturan `True/False` yang fleksibel
Evaluasi lazy (malas)	Data hanya dibaca saat dibutuhkan
Cocok untuk data besar	Tidak perlu load seluruh data sekaligus
Alternatif `filter()`	Bisa dipakai bila menggunakan selector dari iterable, bukan fungsi

Fungsi `compress()` adalah **alat penyaring berbasis iterable Boolean** yang sangat fleksibel, ringan, dan sangat cocok digunakan dalam aplikasi analisis data yang membutuhkan **subset dinamis, acak, atau pola tertentu**.

1.14 Memilih subset dengan islice()

`islice()` adalah fungsi dari modul `itertools` di Python yang digunakan untuk **mengambil sebagian (subset) data dari iterable**, seperti halnya kita menggunakan **slicing (`[:]`) pada list**. Bedanya, `islice()` bisa digunakan pada **semua jenis iterable**, termasuk iterator dan generator, **tanpa perlu mengubahnya jadi list**.

Mengapa `islice()` berguna?

Slicing biasa seperti `data[1::2]` hanya bisa dilakukan pada objek **Sequence** (misalnya list, tuple), dan menghasilkan **list baru** yang bisa memakan banyak memori untuk data besar.

Namun, `islice()`: - Tidak mengubah iterable menjadi list. - Lebih **hemat memori dan efisien**, terutama saat bekerja dengan data besar (file, stream, generator, dsb). - Bisa digunakan pada semua **iterable**, termasuk yang **belum memiliki panjang tetap**.

Cara menggunakan `islice()`

```
from itertools import islice

# Sintaks umum
islice(iterable, start, stop, step)
```

- `start`: posisi awal (default: 0)
- `stop`: posisi akhir (default: sampai habis)
- `step`: langkah (default: 1)

> Sama seperti `range()`, bukan seperti `slice[:]` karena menggunakan **parameter eksplisit**, bukan titik dua.

Contoh Kasus: Membuat pasangan bilangan prima

```
from Chapter04.ch04_ex5 import parse_g

with open("1000.txt") as source:
    flat = list(parse_g(source))  # Daftar bilangan prima

# 10 angka pertama
print(flat[:10])
# -> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

# 10 angka terakhir
print(flat[-10:])
# -> [7841, ..., 7919]
```

Untuk membuat pasangan seperti:
(2, 3), (5, 7), (11, 13), ...

Kita bisa menggunakan:

```
from itertools import islice

flat_iter_1 = iter(flat)
flat_iter_2 = iter(flat)
```

```
pairs = list(zip(
    islice(flat_iter_1, 0, None, 2),  # Ambil angka di indeks genap
    islice(flat_iter_2, 1, None, 2)   # Ambil angka di indeks ganjil
))
```

Hasilnya:

```
pairs[:3]
# -> [(2, 3), (5, 7), (11, 13)]

pairs[-3:]
# -> [(7877, 7879), (7883, 7901), (7907, 7919)]
```

Kenapa pakai 2 iterator?

Setiap `islice()` butuh **iterator mandiri** agar proses slicing tidak saling mengganggu. Kalau kita pakai `islice()` dari satu iterator yang sama, iterator akan **maju terus** dan hasilnya jadi tidak akurat. Maka kita buat:

```
iter_1 = iter(flat)
iter_2 = iter(flat)
```

Pengambilan subset menggunakan `islice()`

Misalnya kita ingin mengambil **1 dari setiap 3 data** dari suatu iterable, maka:

```
from itertools import islice

subset = islice(data, 0, None, 3)
```

sama seperti: `data[0::3]` tetapi: - Bisa digunakan pada iterable (misal: file) - Tidak memakan banyak memori - Cocok untuk **streaming data** besar

Kapan pakai `islice()` daripada slicing `[:]` biasa?

Gunakan `islice()` jika: - Kamu bekerja dengan **data besar** yang tidak muat di memori. - Kamu menggunakan **generator**, **file**, atau **database cursor**. - Kamu butuh performa dan efisiensi.

Perbandingan

Fitur	Slicing `[:]`	`islice()`
Bekerja pada	List/Tuple	Semua Iterable
Hemat memori		
Digunakan untuk streaming		
Bisa digunakan pada generator		

1.15 Penyaringan stateful dengan dropwhile() dan takewhile()

Subtopik ini membahas dua fungsi dari modul `itertools` di Python yang sangat berguna untuk **filtering data secara *stateful***, yaitu `dropwhile()` dan `takewhile()`. Mari kita bahas satu per satu secara sederhana dan menyeluruh agar mudah dipahami.

Stateful berarti fungsi menyimpan **status atau kondisi sementara** saat memproses data. Jadi, filter tidak hanya melihat satu item saja, tapi juga "mengingat" apakah suatu kondisi pernah terpenuhi sebelumnya dan berubah perilaku berdasarkan itu.

1.15.1 `dropwhile()` – *Buang sampai kondisi terpenuhi*

Fungsi `dropwhile(predicate, iterable)` **membuang item dari iterable selama** `predicate` menghasilkan `True`. Begitu `predicate` pertama kali menghasilkan `False`, semua item selanjutnya akan di-*pass through* (dibiarkan lewat).

Analoginya:
Bayangkan Anda membuka buku dan ingin **melewatkan semua halaman awal** sampai Anda melihat kata "BAB 1". Setelah itu, Anda mulai membaca.

Contoh:

```
from itertools import dropwhile

data = [0, 0, 0, 1, 2, 3, 4]
result = list(dropwhile(lambda x: x == 0, data))
print(result)  # [1, 2, 3, 4]
```

1.15.2 `takewhile()` – *Ambil sampai kondisi terpatahkan*

Fungsi `takewhile(predicate, iterable)` **mengambil item dari iterable selama** `predicate` menghasilkan `True`. Begitu `predicate` menghasilkan `False`, fungsi berhenti dan tidak mengambil item lagi.

Analoginya:
Bayangkan Anda hanya ingin membaca *introduction* di buku. Anda mulai dari halaman awal dan berhenti begitu melihat "BAB 1".

Contoh:

```
from itertools import takewhile

data = [1, 2, 3, 0, 4, 5]
result = list(takewhile(lambda x: x != 0, data))
print(result)  # [1, 2, 3]
```

1.15.3 Studi Kasus: Memproses File Warna `crayola.gpl`

Struktur File:

```
GIMP Palette
Name: Crayola
Columns: 16
#
255 73 108\tRadical Red
...
```

File ini punya **header** beberapa baris, diakhiri dengan simbol `#`, lalu diikuti data warna (RGB dan nama warna, dipisahkan tab).

Gunakan `dropwhile()` untuk Lewati Header

Kita ingin melewati semua baris hingga simbol `#` muncul.

Kode:

```
import csv
from pathlib import Path
from itertools import dropwhile, islice

source_path = Path("crayola.gpl")

with source_path.open() as source:
    rdr = csv.reader(source, delimiter='\t')
    row_iter = dropwhile(lambda row: row[0] != '#', rdr)
    color_rows = islice(row_iter, 1, None)
```

Penjelasan: 1. `csv.reader(..., delimiter='\t')`: membaca file dengan delimiter tab (`\t`). 2. `dropwhile(lambda row: row[0] != '#', rdr)`: lewati semua baris **hingga** menemukan baris yang dimulai dengan `#`. 3. `islice(row_iter, 1, None)`: buang baris `#`, lalu ambil semua baris setelahnya, yaitu data warna.

Alternatif: Bedakan Header dan Data Berdasarkan Kolom

Jika semua baris header hanya punya **1 kolom**, sedangkan data warna punya **lebih dari 1 kolom**, bisa juga pakai ini:

```
row_iter = dropwhile(lambda row: len(row) == 1, rdr)
```

Namun ini **kurang fleksibel** karena tidak semua file punya struktur seperti ini.

Ringkasan Perbedaan:

Fungsi	Mode Awal	Berubah Saat	Hasil yang Diambil
`dropwhile()`	Tolak	`predicate == False`	Semua item setelah itu
`takewhile()`	Ambil	`predicate == False`	Semua item sebelum itu

Kapan Dipakai?

- `dropwhile()`: Saat ingin **melewati bagian awal** dari data sampai kondisi berubah.
- `takewhile()`: Saat hanya ingin **mengambil bagian awal** dari data yang memenuhi syarat.

1.16 Dua pendekatan untuk penyaringan dengan filterfalse() dan filter()

Subtopik ini membahas **dua pendekatan filtering** di Python: menggunakan `filter()` dan `filterfalse()` dari modul `itertools`. Keduanya membantu kita **memilih elemen dari iterable berdasarkan fungsi predikat**, tetapi dengan cara yang **berlawanan**.

1.16.1 `filter()` – Pilih yang *benar*

Fungsi `filter(predicate, iterable)` akan menghasilkan hanya elemen yang **memenuhi syarat** (mengembalikan `True` saat dimasukkan ke fungsi `predicate`).

Contoh:

```
data = [0, 1, 2, 3, 4]
result = list(filter(lambda x: x % 2 == 0, data))
print(result)  # [0, 2, 4]
```

1.16.2 `filterfalse()` – Pilih yang *tidak memenuhi*

Fungsi `itertools.filterfalse(predicate, iterable)` melakukan kebalikan dari `filter()` — ia hanya mengembalikan **elemen yang GAGAL memenuhi syarat** (mengembalikan `False` saat diuji dengan fungsi `predicate`).

Contoh:

```
from itertools import filterfalse

data = [0, 1, 2, 3, 4]
result = list(filterfalse(lambda x: x % 2 == 0, data))
print(result)  # [1, 3]
```

1.16.3 Implementasi `filterfalse()` dari `filter()`

Secara konseptual, `filterfalse()` bisa ditulis ulang menggunakan `filter()` seperti ini:

```
filterfalse_concept = lambda pred, iterable: filter(lambda x: not pred(x), iterable)
```

Namun, memakai `filterfalse()` langsung lebih **jelas, aman, dan bisa digunakan ulang** tanpa harus menuliskan negasi (`not`) setiap kali.

1.16.4 `None` sebagai Predicate

Jika `predicate` adalah `None`, maka fungsi akan menyaring berdasarkan **kebenaran nilai itu sendiri**:

Contoh:

```
from itertools import filterfalse

source = [0, False, 1, 2]

print(list(filter(None, source)))         # [1, 2]
print(list(filterfalse(None, source)))    # [0, False]
```

1.16.5 Membagi Data Jadi Dua: Diterima vs Ditolak

Kadang kita ingin memisahkan data ke dalam **dua kelompok**: yang lolos dan yang tidak. Kita bisa melakukannya dengan bantuan `tee()` dari `itertools` untuk menggandakan iterator:

Contoh:

```
from itertools import filterfalse, tee

def rule(x):
    return x > 0
```

```
samples = [1, -2, 3, -4, 0]

iter1, iter2 = tee(iter(samples), 2)
positive = filter(rule, iter1)
non_positive = filterfalse(rule, iter2)

print("Positive:", list(positive))          # [1, 3]
print("Non-positive:", list(non_positive))  # [-2, -4, 0]
```

Kenapa Ini Penting?

- Kita **tidak perlu menulis ulang** fungsi `rule()` atau membuat versi negasinya.
- Ini **menghindari bug logika** yang bisa muncul karena salah tulis `not`, `==`, atau `!=`.
- Kode jadi lebih **bersih, jelas, dan mudah dipelihara**.

1.16.6 Ringkasan:

Fungsi	Mengembalikan…	Cocok Untuk
`filter()`	Item yang *True* untuk rule	Menyaring item yang valid
`filterfalse()`	Item yang *False* untuk rule	Menyaring item yang tidak valid
`filter(None, xs)`	Item yang "truthy"	Menghapus 0, None, False, '', []
`filterfalse(None, xs)`	Item yang "falsy"	Mengambil hanya yang kosong/False/0

1.17 Menerapkan fungsi ke data melalui starmap() dan map()

Subtopik ini menjelaskan **penerapan fungsi ke data menggunakan `map()` dan `starmap()`**, dua pendekatan penting dalam pemrograman fungsional Python.

1.17.1 Konsep Dasar `map()` – Fungsi ke Satu per Satu

Fungsi built-in `map()` akan **menerapkan sebuah fungsi ke setiap elemen dari iterable**. Sederhananya bisa digambarkan seperti ini:

```
map_concept = (
    lambda function, arg_iter: (function(a) for a in arg_iter)
)
```

Contoh sederhana:

```
nums = [1, 2, 3]
squared = list(map(lambda x: x ** 2, nums))
print(squared)  # [1, 4, 9]
```

1.17.2 `starmap()` – Versi *args dari `map()`

`itertools.starmap()` seperti `map()`, tetapi untuk iterable yang berisi **tuple** atau struktur serupa. Ia akan men-*dekomposisi* tuple dan menerapkannya sebagai argumen ke fungsi.

```
from itertools import starmap

starmap_concept = (
    lambda function, arg_iter: (function(*a) for a in arg_iter)
)
```

Contoh:

```
from operator import mul
pairs = [(2, 3), (4, 5), (6, 7)]
products = list(starmap(mul, pairs))
print(products)  # [6, 20, 42]
```

Tanpa `starmap()`, kita harus tulis:

```
[f(*args) for args in pairs]  # Lebih panjang dan mudah salah
```

1.17.3 Studi Kasus: Objek `Leg` dari Data Perjalanan

Dalam contoh praktis, kita punya data GPS dan ingin membentuk objek `LegNT` berisi:

- Titik awal `PointNT`
- Titik akhir `PointNT`
- Jarak antara titik (menggunakan `haversine()`)

```
make_leg = lambda start, end: LegNT(start, end, haversine(start, end))
```

Dengan data pasangan titik (`(start, end)`) dari fungsi `legs()`, kita bisa langsung membentuk daftar perjalanan:

```
trip = list(starmap(make_leg, pair_iter))
```

Alih-alih memanggil `make_leg(*pair)` untuk setiap `pair`, `starmap()` melakukan unpacking untuk kita.

Output contoh:

```
LegNT(
  start=PointNT(latitude=37.549, longitude=-76.330),
  end=PointNT(latitude=37.841, longitude=-76.273),
  distance=17.72
)
```

1.17.4 `map()` juga Bisa Multi-iterable

Saat kita pakai `map()` dengan banyak iterable, ia akan secara otomatis *zip* data dan memanggil fungsi:

```
list(map(f, iter1, iter2))    list(starmap(f, zip(iter1, iter2)))
```

Contoh:

```
def combine(x, y):
    return f"{x}-{y}"
```

```
a = ['a', 'b', 'c']
b = [1, 2, 3]

print(list(map(combine, a, b)))  # ['a-1', 'b-2', 'c-3']
```

Kenapa Gunakan `starmap()`?

Situasi	Gunakan
Iterable berisi tuple	`starmap()`
Iterable tunggal (biasa)	`map()`
Banyak iterable	`map()`
Ingin ringkas dan aman	`starmap()`

1.17.5 Ringkasan

Fungsi	Kegunaan
`map()`	Terapkan fungsi ke setiap elemen (atau beberapa iterable)
`starmap()`	Terapkan fungsi ke iterable-of-tuples, secara unpack otomatis

1.18 Mengkloning iterator dengan `tee()`

`tee()` adalah fungsi dari modul `itertools` yang memungkinkan kita **menggandakan (cloning) sebuah iterator** menjadi beberapa *independent iterators* yang bisa diiterasi secara terpisah **tanpa mengonsumsi data aslinya dua kali**.

Import-nya:

```
from itertools import tee
```

1.18.1 Permasalahan: Iterator Bisa "Habis"

Iterator di Python **tidak bisa diulang** kecuali kita buat ulang:

```
data = iter([1, 2, 3])
print(next(data))  # 1
print(next(data))  # 2

# Tidak bisa kembali ke awal lagi tanpa membuat ulang data
```

Jadi, kalau kamu ingin membaca dari **satu iterator dua kali atau lebih**, kamu perlu *clone* iterator itu. Di sinilah `tee()` membantu!

Cara Kerja `tee()`

`tee(iterable, n)` akan mengembalikan `n` buah **iterator independen** berdasarkan iterable yang diberikan.

Contoh dasar:

```
from itertools import tee

original = iter([1, 2, 3])
it1, it2 = tee(original, 2)

print(list(it1))  # [1, 2, 3]
print(list(it2))  # [1, 2, 3]
```

Setiap iterator akan mengakses data yang sama dari `original`, tapi mereka saling **tidak menganggu**.

Catatan Penting: Gunakan dengan Bijak

- `tee()` **menyimpan buffer internal** untuk memastikan iterator tetap sinkron.
- Jika satu iterator jauh lebih cepat dari yang lain, bisa menyebabkan penggunaan memori yang lebih besar.

Contoh kasus boros memori:

```
a, b = tee(big_iterator)
for x in a:  # kalau a digunakan sepenuhnya dulu
    ...
# b akan butuh buffer besar untuk menyimpan semua nilai agar bisa mulai dari awal
```

Contoh Praktis: Filter dan Filterfalse

Misalnya kamu punya sebuah rule, dan ingin memisahkan data jadi dua kelompok:

```
from itertools import filterfalse, tee

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

nums = range(10)

# Clone iterator agar bisa dipakai dua kali
it1, it2 = tee(nums)
even = filter(is_even, it1)
odd = filterfalse(is_even, it2)

print(list(even))  # [0, 2, 4, 6, 8]
print(list(odd))   # [1, 3, 5, 7, 9]
```

Tanpa `tee()`, kamu harus menyimpan semua data dulu dalam list — tidak efisien untuk data besar atau stream!

1.19 Ringkasan

Hal	Penjelasan
Fungsi	`tee(iterable, n)`
Output	`n` buah iterator independen

Hal	Penjelasan
Kegunaan	Membaca satu iterator beberapa kali
Catatan	Gunakan hati-hati agar tidak boros memori