

# Modul 8 Praktikum Pemrograman Berbasis Fungsi: Modul `functools`

Tujuan praktikum:

1. Mahasiswa mampu memahami penggunaan dekorator cache dan `lru_cache`
2. Mahasiswa mampu menerapkan dekorator total ordering
3. Mahasiswa mampu memahami dan menggunakan fungsi partial untuk statistika
4. Mahasiswa mampu memahami dan menerapkan fungsi reduce untuk pemrosesan data besar

Pada paradigma *functional programming*, fungsi dianggap sebagai *first-class objects*. Artinya, fungsi dalam Python bisa:

- Disimpan dalam variabel
- Dikirim sebagai argumen ke fungsi lain
- Dikembalikan sebagai hasil dari fungsi
- Disimpan dalam struktur data seperti list atau dictionary

Python mendukung paradigma ini secara langsung, dan untuk membantu kita menerapkan pola desain *functional* yang lebih kompleks dan elegan, Python menyediakan modul bawaan yang bernama `functools`.

Modul `functools` berisi berbagai alat bantu (*utilities*) yang mempermudah penerapan teknik-teknik pemrograman fungsional. Modul ini sangat berguna ketika kita ingin menulis kode yang lebih ringkas, efisien, dan bersifat deklaratif.

Pada topik ini, kita akan membahas berbagai **fungsi tingkat tinggi (higher-order functions)** yang disediakan oleh `functools`. Konsep ini melanjutkan pembahasan dari *Higher-Order Functions*, dan akan semakin dikembangkan lagi dalam *Decorator Design Techniques*.

Berikut adalah beberapa fitur utama dari `functools` yang akan kita pelajari:

1. `@cache` dan `@lru_cache` Kedua dekorator ini menyimpan hasil pemanggilan fungsi agar tidak perlu dihitung ulang jika dipanggil dengan argumen yang sama. Sangat berguna untuk mengoptimalkan performa, terutama pada fungsi yang mahal secara komputasi (misalnya perhitungan rekursif atau query berat).
2. `@total_ordering` Dekorator ini memudahkan kita dalam membuat operator perbandingan seperti `<`, `<=`, `>`, `>=` cukup dengan mendefinisikan beberapa metode saja. Ini juga menjadi kesempatan untuk menggabungkan *object-oriented programming* dengan pendekatan fungsional.

3. `partial()` Fungsi ini memungkinkan kita untuk "mengunci" sebagian argumen dari sebuah fungsi, dan membuat fungsi baru dengan parameter yang tersisa. Ini sangat membantu untuk membuat *callback* atau *factory function* yang fleksibel.
4. `reduce()` Merupakan *higher-order function* yang digunakan untuk melakukan operasi reduksi, seperti menjumlahkan elemen list atau menggabungkan nilai secara berurutan. `reduce()` adalah generalisasi dari fungsi `sum()` dan semacamnya.

## Memoisasi dengan Cache: `@cache` dan `@lru_cache`

Python menyediakan dua dekorator dari modul `functools` untuk melakukan *memoization*, yaitu:

- `@lru_cache(maxsize)`
- `@cache` (setara dengan `@lru_cache(None)` tanpa batas ukuran cache)

Memoization adalah teknik untuk menyimpan hasil dari pemanggilan fungsi sehingga tidak perlu menghitung ulang hasil untuk input yang sama. Teknik ini sangat efektif untuk fungsi rekursif atau fungsi dengan komputasi mahal dan argumen yang sering diulang.

### Contoh: Fibonacci dengan `@lru_cache`

```
from functools import lru_cache

@lru_cache(128)
def fibc(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    return fibc(n-1) + fibc(n-2)
```

- Cache menyimpan hingga 128 hasil.
- Jika `fibc(20)` dipanggil berulang kali, hasilnya akan diambil dari cache, bukan dihitung ulang.

### Perbandingan waktu (`timeit`):

- **Tanpa cache:** ~3.23 detik
- **Dengan cache:** ~0.0779 detik

 Jika kita ingin mengukur performa secara adil, kita harus membersihkan cache:

```
fibc.cache_clear()
```

**Contoh Lain: Kombinasi (nCr) dan Faktorial** Formula:  $\binom{p}{r} = \frac{p!}{r!(p-r)!}$

Untuk menghindari komputasi ulang `factorial`, kita bisa cache fungsi `factorial` dari `math`:

```
from functools import cache
from math import factorial

factorial = cache(factorial)
```

#### **Waktu Eksekusi:**

- **Naive factorial:** ~0.174 detik
- **Cached factorial:** ~0.046 detik

#### **Catatan Penting tentang Caching:**

- `@cache` dan `@lru_cache` menyimpan hasil berdasarkan **argumen fungsi**, jadi hanya cocok untuk:
  - Fungsi **murni** (pure functions) — tidak memiliki efek samping.
  - Nilai **immutable** seperti `int`, `str`, `tuple`, dll.
- ❌ Jangan gunakan caching untuk fungsi seperti `print()`:

```
@cache
def myprint(x): print(x)
myprint(5) # hanya mencetak sekali!
```

- Caching bisa **tidak efektif** untuk nilai `float`, karena presisi bisa berbeda-beda sehingga cache jarang cocok.

## Latihan dan Solusi `@cache` dan `@lru_cache`

Latihan 1: Fibonacci dengan rekursi tanpa cache

```
In [1]: from functools import lru_cache
import time

def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

start = time.time()
result = fib(35)
end = time.time()

print(f"Fib(35) = {result}, waktu: {end - start:.4f} detik")
```

Fib(35) = 9227465, waktu: 1.6118 detik

Latihan 2: Optimalkan dengan `lru_cache`

```
In [2]: @lru_cache(maxsize=None)
def fib_cached(n):
    if n < 2:
        return n
    return fib_cached(n-1) + fib_cached(n-2)

start = time.time()
result = fib_cached(35)
end = time.time()

print(f"Fib_cached(35) = {result}, waktu: {end - start:.4f} detik")
```

Fib\_cached(35) = 9227465, waktu: 0.0001 detik

Latihan 3: Hitung faktorial dengan cache terbatas

```
In [3]: @lru_cache(maxsize=3)
def factorial(n):
    print(f"Menghitung factorial({n})")
    if n == 0:
        return 1
    return n * factorial(n-1)

print("Hasil factorial(5):", factorial(5))
print("Cache info:", factorial.cache_info())
print("Panggil lagi factorial(5):", factorial(5))
print("Cache info setelah dipanggil lagi:", factorial.cache_info())
```

Menghitung factorial(5)  
Menghitung factorial(4)  
Menghitung factorial(3)  
Menghitung factorial(2)  
Menghitung factorial(1)  
Menghitung factorial(0)  
Hasil factorial(5): 120  
Cache info: CacheInfo(hits=0, misses=6, maxsize=3, currsize=3)  
Panggil lagi factorial(5): 120  
Cache info setelah dipanggil lagi: CacheInfo(hits=1, misses=6, maxsize=3, currsize=3)

Latihan 4: Gunakan cache untuk fungsi yang sederhana

```
In [4]: from functools import cache

@cache
def square(x):
    print(f"Menghitung square({x})")
    return x * x

print("Hasil square(5):", square(5))
print("Hasil square(10):", square(10))
print("Panggil lagi square(5):", square(5))
```

```
Menghitung square(5)
Hasil square(5): 25
Menghitung square(10)
Hasil square(10): 100
Panggil lagi square(5): 25
```

#### Latihan 5: Fungsi dengan parameter list

```
In [5]: @lru_cache(maxsize=None)
def process_data(data_tuple):
    data = list(data_tuple)
    print("Memproses data...")
    return sum(x * x for x in data)

data = [1, 2, 3, 4, 5]
print("Hasil pertama:", process_data(tuple(data)))
print("Hasil kedua (seharusnya dari cache):", process_data(tuple(data)))
```

```
Memproses data...
Hasil pertama: 55
Hasil kedua (seharusnya dari cache): 55
```

#### Latihan 6: Fungsi dengan parameter keyword

```
In [6]: @lru_cache(maxsize=None)
def greet(name, title="Mr"):
    print(f"Menghasilkan pesan untuk {title} {name}")
    return f"Hello, {title} {name}!"

print(greet("John"))
print(greet("John", title="Dr"))
print(greet("John")) # Ini akan menggunakan cache dari panggilan pertama
```

```
Menghasilkan pesan untuk Mr John
Hello, Mr John!
Menghasilkan pesan untuk Dr John
Hello, Dr John!
Hello, Mr John!
```

#### Latihan 7: Clear cache saat diperlukan

```
In [7]: @lru_cache(maxsize=None)
def expensive_operation(x):
    print(f"Menghitung untuk {x}")
    return x ** 3

print(expensive_operation(3))
print(expensive_operation(3)) # Menggunakan cache
expensive_operation.cache_clear() # Membersihkan cache
print(expensive_operation(3)) # Menghitung Lagi karena cache dibersihkan
```

```
Menghitung untuk 3
27
27
Menghitung untuk 3
27
```

## Latihan 8: Fungsi dengan beberapa parameter

```
In [8]: @lru_cache(maxsize=None)
def kombinasi(a, b, c):
    print(f"Menghitung kombinasi untuk {a}, {b}, {c}")
    return a + b * c

print(kombinasi(1, 2, 3))
print(kombinasi(1, 2, 3)) # Cache hit
print(kombinasi(1, 2, 4)) # Cache miss
print(kombinasi(2, 1, 3)) # Cache miss (urutan parameter penting)
print("Cache info:", kombinasi.cache_info())
```

```
Menghitung kombinasi untuk 1, 2, 3
7
7
Menghitung kombinasi untuk 1, 2, 4
9
Menghitung kombinasi untuk 2, 1, 3
5
Cache info: CacheInfo(hits=1, misses=3, maxsize=None, currsize=3)
```

## Mendefinisikan Objek Class dengan Total Ordering

Pada pemrograman Python, objek dari kelas buatan sendiri sering kali perlu mendukung berbagai **operasi perbandingan** seperti `==`, `<`, `<=`, `>`, dan `>=`. Namun, menulis semua metode ini satu per satu bisa sangat merepotkan dan berpotensi mengandung duplikasi logika.

Python menyediakan solusi yang elegan melalui **dekorator `@total_ordering`** dari modul `functools`. Dekorator ini memungkinkan kita mendefinisikan hanya dua metode perbandingan utama—biasanya `__eq__()` dan `__lt__()`—dan Python akan secara otomatis melengkapi sisanya.

## Kasus Studi: Kartu Remi

Misalnya, kita ingin merepresentasikan kartu remi sebagai objek. Setiap kartu memiliki:

- `rank` (nilai angka, seperti 2 hingga 10, Jack=11, Queen=12, dst.)
- `suit` (jenis: sekop ♠, hati ♥, keriting ♣, wajik ♦)

Kita bisa membuat kelas dengan `NamedTuple` seperti ini:

```
from typing import NamedTuple

class Card1(NamedTuple):
    rank: int
    suit: str
```

Namun, perbandingan antara dua kartu dengan `NamedTuple` akan mempertimbangkan **semua atribut**, termasuk `suit`. Misalnya:

```
>>> c1 = Card1(2, '♠')
>>> c2 = Card1(2, '♥')
>>> c1 == c2
False
```

Dalam beberapa permainan seperti Blackjack atau War, hanya `rank` yang penting, bukan `suit`. Maka perbandingan seperti di atas tidak sesuai dengan logika permainan.

## Solusi: Menggunakan `@total_ordering`

Untuk mengatasi hal ini, kita buat kelas baru `Card2` yang hanya membandingkan `rank`, dan kita tambahkan dekorator `@total_ordering`:

```
from functools import total_ordering
from typing import NamedTuple, Any

@total_ordering
class Card2(NamedTuple):
    rank: int
    suit: str

    def __str__(self) -> str:
        return f"{self.rank:2d}{self.suit}"

    def __eq__(self, other: Any) -> bool:
        match other:
            case Card2():
                return self.rank == other.rank
            case int():
                return self.rank == other
            case _:
                return NotImplemented

    def __lt__(self, other: Any) -> bool:
        match other:
            case Card2():
                return self.rank < other.rank
            case int():
                return self.rank < other
            case _:
                return NotImplemented
```

Penjelasan:

- `__eq__()` dan `__lt__()` adalah metode minimum yang harus didefinisikan.
- `@total_ordering` secara otomatis akan menghasilkan:
  - `__le__()` → `<=`
  - `__gt__()` → `>`

- `__ge__()` → `>=`
- Perbandingan bisa dilakukan antar objek `Card2`, atau antara `Card2` dan `int`.

Contoh penggunaan:

```
>>> c1 = Card2(2, '♠')
>>> c2 = Card2(2, '♥')
>>> c3 = Card2(3, '♦')
>>> c1 == c2
True
>>> c1 < c3
True
>>> c1 == 2
True
>>> c3 > 2
True
```

### Keterbatasan `@total_ordering`

Walaupun `@total_ordering` sangat membantu, ia memiliki keterbatasan: metode yang dihasilkan **tidak pintar dalam menangani tipe campuran dua arah**.

Misalnya:

```
>>> 2 == c1
True
>>> 2 < c3
Traceback (most recent call last):
TypeError: '<' not supported between instances of 'int' and 'Card2'
Mengapa? Karena 2 < c3 akan memanggil int.__lt__(), dan int tidak tahu bagaimana cara membandingkan dirinya dengan Card2.
```

Solusi untuk hal ini adalah dengan menulis semua metode perbandingan (`__le__`, `__gt__`, dll) secara manual agar kita bisa mengatur logika perbandingan dua arah dengan lebih presisi.

### Rekomendasi

- `@total_ordering` sangat cocok untuk **kelas yang memiliki logika perbandingan sederhana dan satu arah**, seperti membandingkan objek dengan nilai numerik internal.
- Untuk **kelas yang mendukung perbandingan kompleks atau dua arah** (misalnya membandingkan `Card2` dengan `int` dari kedua arah), lebih baik menulis semua metode `__lt__`, `__le__`, `__gt__`, dan `__ge__` secara eksplisit.
- Pendekatan ini sangat berguna dalam **simulasi permainan, sistem skor, dan pemrosesan objek semi-numerik**, seperti tanggal, level, bobot, dan tentu saja, kartu remi.

## Latihan dan solusi `@total_ordering`

### Latihan 1: Implementasi kelas tanpa total\_ordering

Buat kelas Mahasiswa yang bisa dibandingkan berdasarkan NIM, dengan mengimplementasikan semua metode perbandingan.

```
In [9]: class Mahasiswa:
    def __init__(self, nim, nama):
        self.nim = nim
        self.nama = nama

    def __eq__(self, other):
        if not isinstance(other, Mahasiswa):
            return NotImplemented
        return self.nim == other.nim

    def __lt__(self, other):
        if not isinstance(other, Mahasiswa):
            return NotImplemented
        return self.nim < other.nim

    def __le__(self, other):
        if not isinstance(other, Mahasiswa):
            return NotImplemented
        return self.nim <= other.nim

    def __gt__(self, other):
        if not isinstance(other, Mahasiswa):
            return NotImplemented
        return self.nim > other.nim

    def __ge__(self, other):
        if not isinstance(other, Mahasiswa):
            return NotImplemented
        return self.nim >= other.nim

m1 = Mahasiswa(123, "Alice")
m2 = Mahasiswa(456, "Bob")
print(m1 < m2) # True
print(m1 >= m2) # False
```

True

False

### Latihan 2: Optimalkan dengan total\_ordering

Gunakan `@total_ordering` untuk menyederhanakan implementasi kelas Mahasiswa.

```
In [10]: from functools import total_ordering

@total_ordering
class Mahasiswa:
    def __init__(self, nim, nama):
        self.nim = nim
        self.nama = nama
```

```

def __eq__(self, other):
    if not isinstance(other, Mahasiswa):
        return NotImplemented
    return self.nim == other.nim

def __lt__(self, other):
    if not isinstance(other, Mahasiswa):
        return NotImplemented
    return self.nim < other.nim

m1 = Mahasiswa(123, "Alice")
m2 = Mahasiswa(456, "Bob")
print(m1 < m2)    # True
print(m1 <= m2)   # True
print(m1 > m2)    # False
print(m1 >= m2)   # False
print(m1 == m2)   # False

```

True  
True  
False  
False  
False

Latihan 3: Kelas dengan multiple atribut untuk perbandingan

Buat kelas Produk yang bisa dibandingkan berdasarkan harga, dan jika harga sama, bandingkan berdasarkan nama.

```

In [11]: @total_ordering
class Produk:
    def __init__(self, nama, harga):
        self.nama = nama
        self.harga = harga

    def __eq__(self, other):
        if not isinstance(other, Produk):
            return NotImplemented
        return (self.harga, self.nama) == (other.harga, other.nama)

    def __lt__(self, other):
        if not isinstance(other, Produk):
            return NotImplemented
        return (self.harga, self.nama) < (other.harga, other.nama)

p1 = Produk("Laptop", 1000)
p2 = Produk("Smartphone", 800)
p3 = Produk("Tablet", 800)

print(p2 < p1)    # True (800 < 1000)
print(p2 < p3)    # True ("Smartphone" < "Tablet" ketika harga sama)
print(p2 <= p3)   # False (karena nama berbeda)

```

True  
True  
True

Latihan 4: Implementasi dengan `__eq__` dan `__gt__`

Tunjukkan bahwa `@total_ordering` bisa bekerja dengan kombinasi metode lain selain `__lt__`.

```
In [12]: @total_ordering
class Titik:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if not isinstance(other, Titik):
            return NotImplemented
        return self.x == other.x and self.y == other.y

    def __gt__(self, other):
        if not isinstance(other, Titik):
            return NotImplemented
        return self.x > other.x or (self.x == other.x and self.y > other.y)

t1 = Titik(1, 2)
t2 = Titik(3, 4)
t3 = Titik(1, 3)

print(t1 < t2)    # True
print(t1 <= t3)   # True
print(t2 > t3)    # True
print(t3 >= t1)   # True
```

```
True
True
True
True
```

## Latihan 5: Kasus edge case dengan None

Handle perbandingan dengan None atau tipe yang tidak sesuai.

```
In [13]: @total_ordering
class Nilai:
    def __init__(self, nilai):
        self.nilai = nilai

    def __eq__(self, other):
        if other is None:
            return False
        if not isinstance(other, Nilai):
            return NotImplemented
        return self.nilai == other.nilai

    def __lt__(self, other):
        if other is None:
            return False # None dianggap lebih kecil dari semua objek
        if not isinstance(other, Nilai):
            return NotImplemented
        return self.nilai < other.nilai
```

```

        return NotImplemented
    return self.nilai < other.nilai

n1 = Nilai(75)
n2 = Nilai(80)
print(n1 < n2)      # True
print(n1 < None)    # False
print(n1 > None)    # True

```

True  
False  
True

## Menerapkan Argumen Parsial dengan `partial()`

Dalam pemrograman, sering kali kita berhadapan dengan fungsi yang menerima banyak argumen. Namun, terkadang beberapa argumen tetap (tidak berubah), dan hanya satu atau dua yang bervariasi. Untuk menangani hal seperti ini dengan elegan, Python menyediakan fungsi `partial()` dari modul `functools`.

Fungsi ini memungkinkan kita untuk membuat **fungsi baru** dari fungsi lama, dengan **beberapa argumen telah ditentukan sebelumnya**. Teknik ini dikenal dengan istilah **partial application** (aplikasi parsial).

- **Partial application** adalah proses membentuk fungsi baru dari fungsi yang sudah ada, dengan **mengunci** (atau **mengikat**) satu atau lebih argumen awal.
- Cara ini sangat berguna untuk menyederhanakan pemanggilan fungsi dan meningkatkan keterbacaan kode.

Contoh Dasar: Eksponensial

Misalkan kita ingin membuat fungsi `2^x`. Kita bisa menggunakan `pow(2, x)` secara langsung, tapi akan terus mengetik angka 2 sebagai basis.

Dengan `partial()`, kita bisa menuliskannya seperti ini:

```

from functools import partial

exp2 = partial(pow, 2)

print(exp2(12))      # Output: 4096 → karena pow(2, 12)
print(exp2(17) - 1) # Output: 131071 → karena pow(2, 17) - 1

```

**Penjelasan:**

- `pow(a, b)` menghitung  $a^b$ .
- Dengan `partial(pow, 2)`, kita membuat fungsi `exp2(x)` yang setara dengan `pow(2, x)`.



### Alternatif: Menggunakan Lambda

Kita juga bisa menulis hal yang sama dengan lambda:

```
exp2 = lambda y: pow(2, y)
```

Kedua pendekatan di atas menghasilkan fungsi yang sama, tetapi:

- `partial()` memberi petunjuk eksplisit bahwa kita sedang melakukan *partial application*.
- `lambda` lebih bebas, tapi kadang **kurang menjelaskan niat desain** program.

## Contoh Nyata: Normalisasi Z-Score

Dalam ilmu data, kita sering menghitung **Z-score** menggunakan rumus:

$$z = (\text{nilai} - \text{mean}) / \text{standar deviasi}$$

Misalkan kita punya fungsi seperti ini:

```
def z(mean: float, stdev: float, score: float) -> float:
    return (score - mean) / stdev
```

Saat kita menghitung z-score dari banyak data yang memiliki **mean** dan **stdev** tetap, kita bisa memanfaatkan `partial()` agar tak perlu menuliskan `mean` dan `stdev` berulang-ulang:

```
from statistics import mean, stdev
from functools import partial

some_data = [12, 15, 14, 10, 13]

m = mean(some_data)
std = stdev(some_data)

z_value = partial(z, m, std) # Membuat fungsi baru dengan mean & std tetap

normalized_some_data = [z_value(x) for x in some_data]
```

### Mengapa Cara ini Berguna?

- Lebih **ringkas** dan **terbaca** daripada harus memanggil `z(m, std, x)` berkali-kali.
- Lebih **jelas secara semantik**: `z_value(x)` menunjukkan bahwa kita sedang menghitung Z-score untuk `x`.

Catatan tentang Urutan Argumen

- `partial()` **mengikat argumen secara posisi**, dari **kiri ke kanan**.
- Fungsi juga bisa menerima **keyword arguments**, seperti:

```
partial(fungsi, arg1, kunci1=nilai)
```

## Hubungan dengan Currying

Meskipun mirip, `partial()` **tidak sama persis** dengan **currying**.

- **Currying** (konsep dari teori fungsi) mengubah fungsi multi-argumen menjadi rantai fungsi satu-argumen.
- **Partial application** (yang didukung oleh `partial()`) hanyalah cara **membuat versi baru dari fungsi** dengan beberapa argumen tetap.

Python tidak secara langsung mendukung currying seperti bahasa fungsional murni (contoh: Haskell), tapi `partial()` memberikan kemampuan yang cukup serupa untuk sebagian besar kebutuhan praktis.

Ringkasan

Fitur	Keterangan
Fungsi <code>partial()</code>	Membuat fungsi baru dari fungsi lama dengan sebagian argumen tetap
Manfaat utama	Menyederhanakan pemanggilan fungsi berulang dengan argumen tetap
Alternatif	Lambda expressions
Kelebihan <code>partial()</code>	Lebih menjelaskan maksud desain, lebih rapi
Kapan digunakan	Saat argumen fungsi ada yang tetap sepanjang waktu, seperti konfigurasi, konteks, atau parameter global

## Latihan dan Solusi `partial()`

### 1. Fungsi Distribusi Normal dengan Parameter Tetap

Latihan: Buat versi khusus dari fungsi normal distribution dengan mean dan std dev yang sudah ditetapkan.

```
In [14]: from functools import partial
import math

def normal_pdf(x, mu=0, sigma=1):
    """Fungsi density distribusi normal"""
    return (1 / (sigma * math.sqrt(2 * math.pi))) * math.exp(-0.5 * ((x - mu) / sig

# Membuat fungsi khusus dengan mu=100 dan sigma=15 (seperti distribusi IQ)
iq_pdf = partial(normal_pdf, mu=100, sigma=15)

# Sekarang kita bisa memanggil iq_pdf hanya dengan x
print(f"Density IQ 115: {iq_pdf(115):.6f}")
print(f"Density IQ 85: {iq_pdf(85):.6f}")
```

```
Density IQ 115: 0.016131
Density IQ 85: 0.016131
```

## 2. Fungsi Statistik dengan Parameter Default

Latihan: Buat versi khusus dari fungsi statistik dengan parameter tertentu.

```
In [15]: def moving_average(data, window_size=3, weights=None):
    """Menghitung moving average dengan optional weights"""
    if weights is None:
        weights = [1/window_size] * window_size
    return sum(x * w for x, w in zip(data[-window_size:], weights))

# Membuat fungsi khusus dengan window_size=5
ma_5 = partial(moving_average, window_size=5)

data = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
print(f"5-point moving average: {ma_5(data)}")

# Membuat versi weighted
weights = [0.1, 0.2, 0.3, 0.4]
ma_weighted = partial(moving_average, weights=weights, window_size=len(weights))
print(f"Weighted moving average: {ma_weighted(data)}")
```

```
5-point moving average: 80.0
Weighted moving average: 90.0
```

## 3. Fungsi Transformasi Data

Latihan: Buat fungsi transformasi data dengan parameter yang sudah ditetapkan.

```
In [16]: def normalize(data, min_val, max_val, new_min=0, new_max=1):
    """Normalisasi data ke range baru"""
    return [(x - min_val) / (max_val - min_val)) * (new_max - new_min) + new_min for x in data]

# Membuat fungsi normalisasi ke range 0-100
normalize_to_100 = partial(normalize, new_min=0, new_max=100)

data = [1, 2, 3, 4, 5]
print(f"Normalized to 0-100: {normalize_to_100(data, min(data), max(data))}")
```

```
Normalized to 0-100: [0.0, 25.0, 50.0, 75.0, 100.0]
```

## 4. Fungsi Hypothesis Testing

Latihan: Buat versi khusus dari fungsi uji hipotesis.

```
In [17]: def confidence_interval(data, confidence=0.95):
    """Menghitung confidence interval"""
    n = len(data)
    mean = sum(data) / n
    stdev = ((sum((x - mean) ** 2 for x in data) / n) ** 0.5
    z = 1.96 # Untuk 95% confidence
```

```
margin = z * (stdev / math.sqrt(n))
return (mean - margin, mean + margin)

# Membuat fungsi khusus untuk 99% confidence
ci_99 = partial(confidence_interval, confidence=0.99)

sample = [180, 172, 178, 185, 190, 195, 192, 200, 210, 190]
print(f"95% CI: {confidence_interval(sample)}")
print(f"99% CI: {ci_99(sample)}")
```

95% CI: (182.65348263578258, 195.7465173642174)  
99% CI: (182.65348263578258, 195.7465173642174)

## 5. Fungsi Pembuatan Plot Statistik

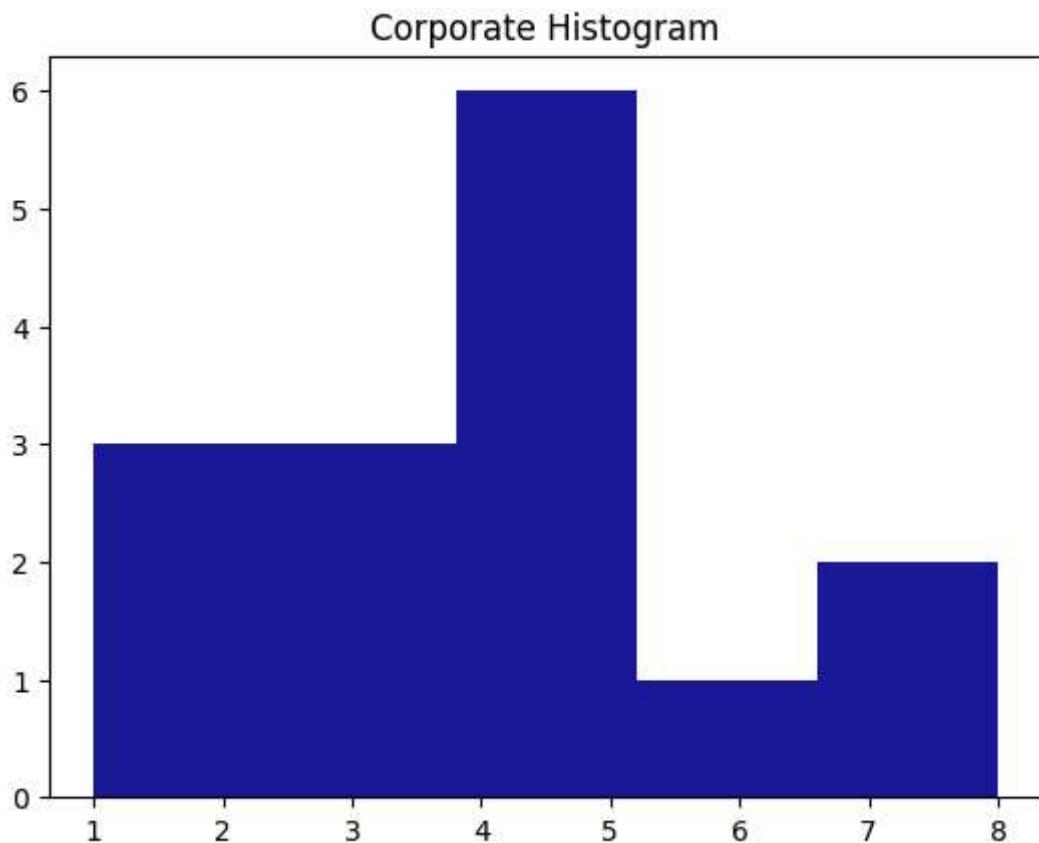
Latihan: Buat fungsi pembuatan plot dengan style default.

```
In [18]: import matplotlib.pyplot as plt

def plot_histogram(data, bins=10, color='blue', alpha=0.7, title='Histogram'):
    """Fungsi untuk membuat histogram"""
    plt.hist(data, bins=bins, color=color, alpha=alpha)
    plt.title(title)
    plt.show()

# Membuat fungsi plot dengan style corporate
corporate_hist = partial(plot_histogram, color='darkblue', alpha=0.9, title='Corporate')

data = [1, 1, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 5, 6, 7, 8]
corporate_hist(data, bins=5) # Hanya perlu tentukan data dan bins
```



## 6. Fungsi Regresi Linear

Latihan: Buat fungsi prediksi regresi dengan parameter model yang sudah ditraining.

```
In [19]: def linear_predict(x, slope, intercept):
    """Fungsi prediksi linear sederhana"""
    return slope * x + intercept

# Setelah training model dengan slope=2.5 dan intercept=10
trained_model = partial(linear_predict, slope=2.5, intercept=10)

# Prediksi untuk nilai x baru
print(f"Prediksi untuk x=5: {trained_model(5)}")
print(f"Prediksi untuk x=10: {trained_model(10)}")
```

Prediksi untuk x=5: 22.5

Prediksi untuk x=10: 35.0

## Menurangi Sekumpulan Data dengan Fungsi `reduce()`

Fungsi-fungsi seperti `sum()`, `len()`, `max()`, dan `min()` sebenarnya adalah bentuk spesifik dari algoritma umum yang bisa diekspresikan dengan fungsi `reduce()`. Fungsi `reduce()` adalah **higher-order function** yang menggabungkan elemen-elemen dalam iterable menggunakan **operasi biner**, satu per satu, dari kiri ke kanan.

Fungsi `reduce()` digunakan untuk mereduksi (menggabungkan) seluruh elemen dalam sebuah iterable menjadi satu nilai, menggunakan sebuah fungsi dua argumen.

Sintaks:

```
reduce(function, iterable, initial)
```

- `function` : fungsi penggabungan dua nilai.
- `iterable` : data yang akan direduksi.
- `initial` : nilai awal yang disediakan sebelum proses reduksi dimulai (opsional)

Contoh:

```
from functools import reduce

d = [2, 4, 4, 4, 5, 5, 7, 9]
hasil = reduce(lambda x, y: x + y, d) # Hasilnya: 40
Ini setara dengan 2+4+4+4+5+5+7+9 .
```

### Evaluasi dari Kiri ke Kanan (Left Fold)

Python mengevaluasi operator dari **kiri ke kanan**, sehingga hasil `reduce()` akan sama dengan evaluasi biasa. Tidak ada opsi *fold right* (evaluasi dari kanan ke kiri) seperti pada bahasa fungsional seperti Haskell atau OCaml.

### Nilai Awal (Initial Value)

Kita bisa memberikan nilai awal pada `reduce()`:

```
reduce(lambda x, y: x + y**2, d, 0) # Hasil: 232
```

Tanpa nilai awal, `reduce()` akan menggunakan elemen pertama, **tanpa menerapkan transformasi** padanya — ini bisa menyebabkan hasil salah.

Misalnya:

```
reduce(lambda x, y: x + y**2, d) # Salah: 230
# Karena dihitung sebagai: 2 + 42 + 42 + ... bukan 0 + 22 + 42 + ...
```

## Definisi Reduksi Umum

Beberapa fungsi `reduce()` dapat didefinisikan seperti berikut:

```
from functools import reduce
from typing import cast, Callable, TypeAlias

FloatFT: TypeAlias = Callable[[float, float], float]

# Penjumlahan kuadrat
sum2 = lambda data: reduce(cast(FloatFT, lambda x, y: x + y**2), data, 0.0)
```

```

# Penjumlahan biasa
sum = lambda data: reduce(cast(FloatFT, lambda x, y: x + y), data, 0.0)

# Menghitung jumlah elemen (seperti len())
count = lambda data: reduce(cast(FloatFT, lambda x, y: x + 1), data, 0.0)

# Nilai minimum
min = lambda data: reduce(cast(FloatFT, lambda x, y: x if x < y else y),
                           data)

# Nilai maksimum
max = lambda data: reduce(cast(FloatFT, lambda x, y: x if x > y else y),
                           data)

```

Catatan: Fungsi `cast()` digunakan untuk memberitahu `mypy` bahwa `lambda` menerima dua `float` dan mengembalikan `float`. Tanpa ini, tipe default-nya adalah `Any`.

## Catatan Desain

- Meskipun `lambda` singkat, **fungsi yang didefinisikan lengkap** lebih baik karena:
  - Bisa memiliki **nilai default**
  - Dapat diberi **dokumentasi**
  - Bisa diuji dengan **doctest**
- Jadi, jika fungsi `lambda` terlalu rumit atau penting, ubahlah jadi fungsi biasa.

## Menggabungkan `map()` dan `reduce()`

Python memungkinkan penggunaan gabungan fungsi `map()` dan `reduce()` untuk memproses koleksi data secara fungsional. Kombinasi ini sangat umum digunakan dalam pemrosesan data besar (map-reduce paradigm), terutama dalam konteks seperti pemrograman paralel atau fungsional.

### Definisi Fungsi `map_reduce()`

```

from collections.abc import Callable, Iterable
from functools import reduce
from typing import TypeVar, cast

ST = TypeVar("ST")

def map_reduce(
    map_fun: Callable[[ST], float],
    reduce_fun: Callable[[float, float], float],
    source: Iterable[ST],
    initial: float = 0
) -> float:
    return reduce(reduce_fun, map(map_fun, source), initial)

```

### Penjelasan:

- `source` : iterable berisi elemen bertipe `ST` (generic).
- `map_fun` : fungsi yang mengubah elemen `ST` menjadi `float`.
- `reduce_fun` : fungsi yang menggabungkan dua `float` menjadi satu nilai akhir.
- `initial` : nilai awal untuk `reduce()`.

Contoh: Menjumlahkan Kuadrat Angka

```
def sum2_mr(source_iter: Iterable[float]) -> float:
    return map_reduce(
        map_fun=lambda y: y**2,
        reduce_fun=lambda x, y: x + y,
        source=source_iter,
        initial=0
    )
```

### Penjelasan:

- `map_fun` : mengubah setiap elemen menjadi kuadratnya.
- `reduce_fun` : menjumlahkan elemen-elemen tersebut.

Contoh penggunaan:

```
>>> sum2_mr([2, 4, 5])
45 # karena  $2^2 + 4^2 + 5^2 = 4 + 16 + 25 = 45$ 
```

⚠ Catatan Meskipun powerful, penggunaan `reduce()` perlu hati-hati karena:

- Tidak semua operasi bisa direduksi dengan aman (khususnya yang tidak asosiatif).
- Tanpa nilai awal (`initial`), hasil bisa salah seperti dijelaskan sebelumnya.
- `lambda` terlalu ringkas kadang menyulitkan alat analisis tipe seperti `mypy`.

**Memahami cara penggunaan `groupby()`,  
`defaultdict`, dan `reduce` untuk menganalisis  
data yang dikelompokkan berdasarkan `key`.**

**Data Awal** Data terdiri dari pasangan `(key, value)` seperti ini:

```
data = [
    ('4', 6.1), ('1', 4.0), ('2', 8.3), ('2', 6.5),
    ('1', 4.6), ('2', 6.8), ('3', 9.3), ('2', 7.8),
    ('2', 9.2), ('4', 5.6), ('3', 10.5), ('1', 5.8),
    ('4', 3.8), ('3', 8.1), ('3', 8.0), ('1', 6.9),
    ('3', 6.9), ('4', 6.2), ('1', 5.4), ('4', 5.8)
]
```

### Tujuan

1. Kelompokkan data berdasarkan nilai pertama dari tuple (`'1'`, `'2'`, dst).

2. Hitung **mean** dan **variansi** nilai `float` dalam setiap grup.

3. Bandingkan statistik dari setiap grup.

**Fungsi `partition()`** Fungsi ini menggunakan `defaultdict(list)` untuk mengelompokkan data berdasarkan key.

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

DT = TypeVar("DT")
KT = TypeVar("KT")

def partition(
    source: Iterable[DT],
    key: Callable[[DT], KT]
) -> Iterator[tuple[KT, Iterator[DT]]]:
    pd: dict[KT, list[DT]] = defaultdict(list)
    for item in source:
        pd[key(item)].append(item)
    for k in sorted(pd):
        yield k, iter(pd[k])
```

### Contoh Pemanggilan

```
for key, group_iter in partition(data, key=lambda x: x[0]):
    print(key, tuple(group_iter))
```

Hasilnya:

```
1 (('1', 4.0), ('1', 4.6), ('1', 5.8), ('1', 6.9), ('1', 5.4))
2 (('2', 8.3), ('2', 6.5), ('2', 6.8), ('2', 7.8), ('2', 9.2))
3 (('3', 9.3), ('3', 10.5), ('3', 8.1), ('3', 8.0), ('3', 6.9))
4 (('4', 6.1), ('4', 5.6), ('4', 3.8), ('4', 6.2), ('4', 5.8))
```

**Fungsi `summarize()`** Digunakan untuk menghitung mean dan variance dari nilai dalam setiap grup.

```
from typing import Sequence

def summarize(
    key: KT,
    item_iter: Iterable[tuple[KT, float]]
) -> tuple[KT, float, float]:
    def mean(seq: Sequence[float]) -> float:
        return sum(seq) / len(seq)

    def var(mean: float, seq: Sequence[float]) -> float:
        return sum((x - mean) ** 2 / (len(seq) - 1) for x in seq)

    values = tuple(v for k, v in item_iter)
    m = mean(values)
    return key, m, var(m, values)
```

### Menggabungkan dengan `starmap()`

```

from itertools import starmap

# Menggunakan partition() versi defaultdict
groups1 = starmap(summarize, partition(data, key=lambda x: x[0]))

# Atau menggunakan partition_s() versi itertools.groupby()
from itertools import groupby
def partition_s(source, key):
    return groupby(sorted(source, key=key), key)

groups2 = starmap(summarize, partition_s(data, key=lambda x: x[0]))

```

**Hasil Akhir**

```

plaintext
1 5.34 1.25
2 7.72 1.22
3 8.56 1.9
4 5.5 0.96

```

- Nilai pertama: key grup
- Nilai kedua: **rata-rata**
- Nilai ketiga: **varian**

### Menggunakan `reduce()` untuk merangkum data

Kita bisa menambahkan penggunaan `reduce()` dari modul `functools` untuk menunjukkan bagaimana kita bisa melakukan agregasi data (misalnya total nilai, rata-rata, atau akumulasi hasil) secara manual. Dalam konteks data yang telah dikelompokkan, `reduce()` bisa dipakai sebagai alternatif pendekatan terhadap ringkasan statistik seperti jumlah total, rata-rata, atau bahkan kombinasi nilai.

Mari kita lengkapi contoh `summarize()` menggunakan `reduce()` untuk menghitung **Jumlah dan jumlah kuadrat** (yang bisa digunakan untuk rata-rata dan varians).

```

from functools import reduce
from typing import Iterable, Tuple

def summarize_with_reduce(
    key: KT,
    item_iter: Iterable[tuple[KT, float]]
) -> tuple[KT, float, float]:
    def reducer(acc: tuple[float, float, int], item: tuple[KT, float]) ->
        tuple[float, float, int]:
            _, value = item
            total, total_sq, count = acc
            return total + value, total_sq + value ** 2, count + 1

    total, total_sq, count = reduce(reducer, item_iter, (0.0, 0.0, 0))
    if count <= 1:
        return key, total / count if count > 0 else 0.0, 0.0

```

```

mean = total / count
variance = (total_sq - count * mean ** 2) / (count - 1)
return key, mean, variance

```

Contoh Penggunaan

```

from itertools import starmap

partitioned = partition(data, key=lambda x: x[0])
summary_results = starmap(summarize_with_reduce, partitioned)

for group in summary_results:
    print(group)

```

Penjelasan Singkat

- `reducer()` menerima akumulator (`total, total_sq, count`) dan item ('key', `value`).
- Mengakumulasi nilai (`total`) dan kuadratnya (`total_sq`) serta menghitung jumlah data (`count`).
- Dengan rumus variansi statistik klasik:

$$\text{variance} = \frac{\sum x^2 - n \cdot \bar{x}^2}{n-1}$$

Kapan `reduce()` dapat digunakan?

`reduce()` berguna ketika:

- Kita ingin menghindari membangun struktur data baru (seperti `list` atau `tuple`) di memori.
- Ingin mengontrol secara eksplisit proses akumulasi.
- Menyukai pendekatan functional programming.

## Latihan dan Solusi Reduce dan kombinasi

### 1. Menghitung Total (Sum)

Contoh paling sederhana - menjumlahkan semua elemen:

```
In [20]: from functools import reduce

data = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, data)
print(f"Total: {total}") # Output: 15

# Dengan initializer
total = reduce(lambda x, y: x + y, data, 10)
print(f"Total dengan initial 10: {total}") # Output: 25
```

```
Total: 15
Total dengan initial 10: 25
```

## 2. Menghitung Produk (Perkalian)

Menghitung hasil perkalian semua elemen:

```
In [21]: numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(f"Produk: {product}") # Output: 120

# Faktorial
n = 5
factorial = reduce(lambda x, y: x * y, range(1, n+1))
print(f"Faktorial {n}: {factorial}") # Output: 120
```

```
Produk: 120
Faktorial 5: 120
```

## 3. Mencari Nilai Maksimum

Implementasi alternatif untuk mencari nilai maksimum:

```
In [22]: data = [42, 13, 56, 7, 99, 32]
max_value = reduce(lambda x, y: x if x > y else y, data)
print(f"Nilai maksimum: {max_value}") # Output: 99
```

```
Nilai maksimum: 99
```

## 4. Menggabungkan String

Menggabungkan list string menjadi satu string:

```
In [23]: words = ["Python", "itu", "menyenangkan", "!"]
sentence = reduce(lambda x, y: f"{x} {y}", words)
print(f"Kalimat: {sentence}") # Output: "Python itu menyenangkan !"
```

```
Kalimat: Python itu menyenangkan !
```

## 5. Operasi pada List of Dictionaries

Menghitung total nilai dari list dictionaries:

```
In [24]: transactions = [
    {'amount': 100, 'currency': 'IDR'},
    {'amount': 200, 'currency': 'IDR'},
    {'amount': 50, 'currency': 'USD'},
    {'amount': 150, 'currency': 'IDR'}
]

# Hanya jumlahkan yang currency IDR
total_idr = reduce(
```

```

lambda acc, val: acc + val['amount'] if val['currency'] == 'IDR' else acc,
transactions,
0 # Initializer
)
print(f"Total transaksi IDR: {total_idr}") # Output: 450

```

Total transaksi IDR: 450

## 6. Flatten List

Menggabungkan list bersarang (nested lists):

```

In [25]: nested_lists = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
flattened = reduce(lambda x, y: x + y, nested_lists)
print(f"List rata: {flattened}") # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
List rata: [1, 2, 3, 4, 5, 6, 7, 8, 9]

```

## 7. Komposisi Fungsi

Menggabungkan beberapa fungsi menjadi satu:

```

In [26]: def compose(*functions):
    return reduce(lambda f, g: lambda x: f(g(x)), functions)

def add_5(x):
    return x + 5

def multiply_3(x):
    return x * 3

def square(x):
    return x ** 2

# Membuat fungsi komposit: square -> multiply_3 -> add_5
composed_func = compose(add_5, multiply_3, square)

result = composed_func(4) # (4^2)*3 +5 = 16*3+5 = 48+5 = 53
print(f"Hasil komposisi fungsi: {result}") # Output: 53

```

Hasil komposisi fungsi: 53

## 8. Menghitung Rata-rata

Contoh lebih kompleks dengan menyimpan state:

```

In [27]: data = [1, 2, 3, 4, 5]

def avg_reducer(acc, val):
    # acc adalah tuple (total, count)
    total, count = acc
    return (total + val, count + 1)

```

```
total_sum, total_count = reduce(avg_reducer, data, (0, 0))
average = total_sum / total_count
print(f"Rata-rata: {average}") # Output: 3.0
```

Rata-rata: 3.0

## 9. Map-Reduce Sederhana (Word Count)

```
In [28]: from functools import reduce

words = ["hello", "world", "hello", "python", "world", "hello"]

# Map: Convert each word to (word, 1)
mapped = map(lambda word: (word, 1), words)

# Reduce: Sum the counts for each word
def reducer(acc, item):
    word, count = item
    acc[word] = acc.get(word, 0) + count
    return acc

word_counts = reduce(reducer, mapped, {})
print(word_counts) # Output: {'hello': 3, 'world': 2, 'python': 1}

{'hello': 3, 'world': 2, 'python': 1}
```

## 10. Map-Reduce dengan partial

```
In [29]: from functools import reduce, partial

# Fungsi dengan parameter tambahan
def power(x, exponent):
    return x ** exponent

# Menggunakan partial untuk membuat fungsi square
square = partial(power, exponent=2)

numbers = [1, 2, 3, 4, 5]
squared = map(square, numbers)
sum_of_squares = reduce(lambda x, y: x + y, squared)

print(f"Sum of squares: {sum_of_squares}") # Output: 55
```

Sum of squares: 55

## 11. Kombinasi filter-map-reduce

```
In [30]: from functools import reduce

data = [15, 23, 8, 42, 4, 16, 19]

# Filter angka genap, map kuadrat, reduce untuk menjumlahkan
result = reduce(
    lambda x, y: x + y,
```

```

    map(
        lambda x: x**2,
        filter(lambda x: x % 2 == 0, data)
    ),
    0
)

print(f"Sum of squares of even numbers: {result}") # Output: 82 + 422 + 42 + 162 =

```

Sum of squares of even numbers: 2100

## 12. Map-Reduce dengan lru\_cache

```

In [31]: from functools import reduce, lru_cache

@lru_cache(maxsize=None)
def expensive_computation(x):
    print(f"Computing for {x}...")
    return x * x # Simulasi komputasi berat

numbers = [1, 2, 3, 2, 1, 3, 4]

# Map dengan caching
mapped = map(expensive_computation, numbers)

# Reduce untuk mengalikan hasil
product = reduce(lambda x, y: x * y, mapped)

print(f"Product of computations: {product}")
# Output akan menunjukkan "Computing for..." hanya untuk nilai unik

```

Computing for 1...  
 Computing for 2...  
 Computing for 3...  
 Computing for 4...  
 Product of computations: 20736

## 13. Pipeline dengan reduce dan Fungsi Parsial

```

In [32]: from functools import reduce, partial

def pipeline(data, functions):
    return reduce(lambda x, func: func(x), functions, data)

# Fungsi-fungsi transformasi
square = partial(map, lambda x: x**2)
filter_even = partial(filter, lambda x: x % 2 == 0)
sum_all = partial(reduce, lambda x, y: x + y)

data = [1, 2, 3, 4, 5, 6]

result = pipeline(data, [
    square,
    filter_even,
    sum_all
])

```

```
])
```

```
print(f"Sum of squares of even numbers: {result}") # Output: 4 + 16 + 36 = 56
```

```
Sum of squares of even numbers: 56
```