Modul_6_PBF

April 14, 2025

1 Modul 6 Praktikum Pemrograman Berbasis Fungsi: Objek Kompleks Tak Berstatus

1.1 Tujuan Praktikum:

1. Mahasiswa Memahami Penggunaan Objek Kompleks Tak Berstatus dengan NamedTuple
2. Mahasiswa Memahami Penggunaan Pustaka Pyrsistent
3. Mahasiswa Dapat Merancang Solusi yang Fungsional untuk Representasi Data Kompleks

1.2 Pendahuluan

Stateless Complex Object dalam konteks Python berarti objek yang tidak memiliki status internal yang berubah (immutable), dan biasanya digunakan untuk merepresentasikan data kompleks, seperti data terstruktur (tuple dalam tuple), tanpa perilaku atau metode.

Mengapa digunakan? 1. Lebih aman karena immutable 2. Mudah untuk diuji dan diprediksi 3. Cocok untuk functional programming 4. Digunakan di banyak struktur data (graph, tree, dsb)

Tujuan Penggunaan Object Stateless

1. Meningkatkan keterbacaan dan kejelasan struktur data
2. Memudahkan pengujian unit karena tidak memiliki efek samping
3. Selaras dengan paradigma fungsional Python modern
4. Bisa digunakan sebagai alternatif terhadap class OOP biasa

Teknik Penyelesaian dengan Objek Kompleks Tak Berstatus

1. Gunakan lambda (atau fungsi dari def) untuk memilih item berdasarkan indeks
2. Gunakan lambda (atau def) dengan beberapa parameter posisi + *args
3. Gunakan NamedTuple untuk memilih item dengan nama atribut atau indeks
4. Gunakan @dataclass(frozen=True)
5. Gunakan Pustaka pyrsistent

1.3 Nested Tuple dengan lambda atau def biasa

Kita bisa membuat dan mengelola data nested tuple secara stateless dengan fungsi biasa (def) atau lambda.

[1]:
```
data = ("Alice", ("Math", 90), ("Science", 85))

def get_subject_score(data, subject):
```

```
    if data[1][0] == subject:
        return data[1][1]
    elif data[2][0] == subject:
        return data[2][1]
    return None

print(get_subject_score(data, "Science"))  # Output: 85
```

```
85
```

[2]:
```
# versi lambda

get_math_score = lambda d: d[1][1] if d[1][0] == "Math" else None
print(get_math_score(data))  # Output: 90
```

```
90
```

1.4 NamedTuple dari typing

NamedTuple dari modul typing adalah cara modern (sejak Python 3.6) untuk membuat struktur data yang terlihat seperti class tapi: 1. Tetap immutable 2. Tetap ringan (stateless) 3. Mendukung type hints 4. Terbaca seperti class OOP 5. Bisa punya default values dan docstring

Membangun objek kompleks tak berstatus dengan NamedTuple bertujuan untuk membuat object yang bersifat seperti tuple, tapi punya nama atribut yang jelas (.field bukan hanya [0]), dan bisa digunakan seperti struktur data kecil yang representatif dan aman.

Modul typing

Python, yang secara tradisional dikenal sebagai bahasa dengan **dynamic typing**, memasuki era baru dalam hal kejelasan kode dan pencegahan bug dengan diperkenalkannya modul `typing`. Awalnya hadir sebagai **backport** untuk versi sebelum Python 3.5, modul ini resmi menjadi bagian dari pustaka standar sejak Python 3.5 melalui **PEP 484**. Kehadiran `typing` menandai perubahan penting dalam ekosistem Python, karena memungkinkan pengembang untuk menambahkan **type hint** pada kode mereka. Dengan begitu, Python tetap mempertahankan fleksibilitasnya, namun juga mendapatkan manfaat dari **static typing** — seperti peningkatan keterbacaan, dokumentasi otomatis, dan kemampuan deteksi kesalahan lebih awal melalui alat seperti `mypy` atau `pyright`.

Sejak diperkenalkan, modul `typing` terus mengalami perkembangan, dengan berbagai fitur dan tipe baru yang ditambahkan dalam versi-versi Python selanjutnya. Sebelum Python 3.5, penambahan tipe bersifat opsional dan hanya bisa dilakukan dengan pustaka eksternal. Kini, dengan dukungan penuh dari bahasa inti, `typing` memungkinkan developer untuk **menjelaskan maksud dan kontrak fungsi secara eksplisit** melalui anotasi tipe. Modul ini membawa bentuk **pemeriksaan tipe statis (static type checking)** ke dalam bahasa yang selama ini sangat dinamis, dan memberikan kontribusi besar terhadap peningkatan kualitas kode Python yang besar dan kompleks.

Contoh Dasar:

[ ]:
```
from typing import NamedTuple
```

```python
class Point(NamedTuple):
    x: float
    y: float

p = Point(1.5, 2.5)
print(p.x)  # 1.5
print(p)    # Point(x=1.5, y=2.5)

# Point dibuat seperti class biasa
# Tapi sebenarnya adalah subclass dari tuple
# Atribut tidak bisa diubah: p.x = 3  akan error
```

```
1.5
Point(x=1.5, y=2.5)
```

Contoh lainnya

[5]:
```python
class Point(NamedTuple):
    latitude: float
    longitude: float

class Leg(NamedTuple):
    start: Point
    end: Point
    distance: float

trip = Leg(Point(0.0, 100.0), Point(5.0, 101.0), 150.5)

print(trip.start.latitude)  # 0.0
print(trip.distance)        # 150.5
print(trip)
```

```
0.0
150.5
Leg(start=Point(latitude=0.0, longitude=100.0), end=Point(latitude=5.0,
longitude=101.0), distance=150.5)
```

Default value bisa ditambahkan seperti class biasa — sejak Python 3.7

[6]:
```python
class User(NamedTuple):
    username: str
    level: int = 1

u = User("alice")
print(u)  # User(username='alice', level=1)
```

```
User(username='alice', level=1)
```

Contoh Latihan 1. Buat class MahasiswaNT yang menyimpan nama dan ipk. Cetak nama.

[7]:
```python
from typing import NamedTuple

class MahasiswaNT(NamedTuple):
    nama: str
    ipk: float

m = MahasiswaNT("Rina", 3.85)
print(m.nama)
```

```
Rina
```

2. Buat class BukuNT dengan judul, penulis, dan tahun. Cetak penulis.

[8]:
```python
class BukuNT(NamedTuple):
    judul: str
    penulis: str
    tahun: int

b = BukuNT("Sapiens", "Yuval Noah Harari", 2014)
print(b.penulis)
```

```
Yuval Noah Harari
```

3. Buat struktur PerjalananNT yang memiliki:
   - asal: PointNT
   - tujuan: PointNT
   - jarak: float
   - durasi: float Lalu, cetak kota asal dan jaraknya.

[9]:
```python
class PointNT(NamedTuple):
    nama_kota: str
    latitude: float
    longitude: float

class PerjalananNT(NamedTuple):
    asal: PointNT
    tujuan: PointNT
    jarak: float
    durasi: float

asal = PointNT("Bandar Lampung", -5.397, 105.267)
tujuan = PointNT("Palembang", -2.976, 104.775)

perjalanan = PerjalananNT(asal, tujuan, 325.5, 6.5)

print(f"Perjalanan dari {perjalanan.asal.nama_kota} ke {perjalanan.tujuan.
  ↪nama_kota}")
print(f"Jarak: {perjalanan.jarak} km")
```

```
Perjalanan dari Bandar Lampung ke Palembang
```

```
Jarak: 325.5 km
```

`NamedTuple` dengan Method

`NamedTuple` dari modul `typing` di Python merupakan bentuk evolusi dari struktur `tuple` biasa, yang dilengkapi dengan **nama atribut** dan **type hinting**, serta mendukung penambahan **method (opsional)** seperti pada class biasa. Dapat dikatakan bahwa `NamedTuple` adalah kombinasi dari **Tuple + Nama Atribut + Type Hint + Method**, menjadikannya pilihan yang tepat ketika kita membutuhkan struktur data ringan yang bersifat immutable, tetapi juga ingin menyematkan logika atau perilaku tambahan melalui method.

Karena `NamedTuple` tetap turunan dari `tuple`, fungsi ini mempertahankan sifat-sifat `tuple` seperti ketidakberubahannya (immutable), tetapi dengan peningkatan dalam keterbacaan dan dokumentasi kode. Kita bisa menambahkan method di dalam class `NamedTuple` seperti layaknya pada class Python biasa, yang memungkinkan kita menyisipkan **fungsi analisis, perhitungan, validasi, atau representasi** tanpa kehilangan efisiensi struktur data.

Pada konteks statistik atau analisis data, `NamedTuple` dapat dipakai untuk mewakili entitas seperti titik data, hasil pengukuran, atau statistik ringkasan. Misalnya, kita bisa membuat method untuk menghitung rata-rata, deviasi, atau transformasi data langsung dari objek tersebut. Method-method lain yang umum ditambahkan adalah `distance`, `as_dict()`, atau `summary()` tergantung pada kebutuhan.

Contoh Penggunaan `NamedTuple` dengan Method

[10]:
```python
from typing import NamedTuple
from math import sqrt

class DataPoint(NamedTuple):
    x: float
    y: float

    def magnitude(self) -> float:
        return sqrt(self.x**2 + self.y**2)

    def scale(self, factor: float) -> "DataPoint":
        return DataPoint(self.x * factor, self.y * factor)
```

Contoh di atas menunjukkan `DataPoint`, yaitu titik dua dimensi dengan method `magnitude()` untuk menghitung panjang vektor dari titik ke origin, dan `scale()` untuk mengalikan koordinat dengan skalar tertentu. Cara ini sangat berguna dalam aplikasi statistik, fisika, atau machine learning yang menggunakan operasi vektor.

[11]:
```python
class StatistikRingan(NamedTuple):
    min_val: float
    max_val: float
    mean: float

    def rentang(self) -> float:
        return self.max_val - self.min_val
```

Contoh lain dalam konteks statistik:

Dengan struktur ini, kita bisa dengan mudah membuat objek hasil perhitungan statistik dan tetap bisa memanggil method seperti `.rentang()` untuk mendapatkan informasi tambahan.

`NamedTuple` tidak hanya bermanfaat sebagai representasi data yang ringkas dan aman, tetapi juga dapat diperkaya dengan method seperti class biasa. Ini menjadikannya pilihan tepat ketika kita ingin kombinasi antara **struktur data ringan, aman, dapat dibaca**, dan tetap **bisa digunakan secara fungsional maupun prosedural**. Method-method yang umum digunakan di dalam `NamedTuple` meliputi operasi matematika, konversi data, analisis, validasi, serta representasi khusus untuk laporan atau log.

NamedTuple untuk Representasi Graf

Pada sebuah graf, kita sering berhadapan dengan struktur seperti: - Node (simpul): Biasanya berisi ID dan atribut tambahan seperti label atau bobot. - Edge (sisi): Menghubungkan dua node dan bisa memiliki bobot. - Path (jalur): Urutan edge atau node yang dilewati.

[ ]:
```python
from typing import NamedTuple, List

# Node menyimpan identitas dan label.
class Node(NamedTuple):
    id: str
    label: str

# Edge mewakili sisi dari node sumber ke node tujuan,
# dengan bobot yang menyatakan jarak atau biaya.
class Edge(NamedTuple):
    source: Node
    target: Node
    weight: float

    # Lengkap dengan method is_loop() untuk cek apakah sisi tersebut␣
 ↪menghubungkan node ke dirinya sendiri.
    def is_loop(self) -> bool:
        return self.source.id == self.target.id

# Path adalah kumpulan edge, dengan method total_weight() untuk menghitung␣
 ↪total bobot dari jalur.
class Path(NamedTuple):
    edges: List[Edge]

    def total_weight(self) -> float:
        return sum(edge.weight for edge in self.edges)
```

[13]:
```python
A = Node(id="A", label="Start")
B = Node(id="B", label="Middle")
C = Node(id="C", label="End")
```

```
e1 = Edge(source=A, target=B, weight=2.5)
e2 = Edge(source=B, target=C, weight=3.0)
e3 = Edge(source=C, target=C, weight=1.0)

path = Path(edges=[e1, e2])

print(f"Is e3 a loop? {e3.is_loop()}")
print(f"Total weight of path: {path.total_weight()}")
print(f"Path edges: {[f'{edge.source.id}->{edge.target.id}' for edge in path.
  ↪edges]}")
```

```
Is e3 a loop? True
Total weight of path: 5.5
Path edges: ['A->B', 'B->C']
```

1.4.1 Studi kasus: Weather Tracker

Kamu sedang melacak suhu harian di kotamu selama satu minggu melalui situs suhu harian. Data ini terekam dalam format berikut:

```
# Sample data: Riwayat Suhu Harian
data = {
    "day": ["Senin", "Selasa", "Rabu", "Kamis", "Jumat", "Sabtu", "Minggu"],
    "temperature": [28.5, 30.0, 29.2, 27.8, 31.0, 30.5, 28.0]
}
```

1. Hari apa terjadi kenaikan suhu tertinggi dibanding hari sebelumnya?
2. Hari mana terjadi penurunan suhu terbesar?
3. Berapa rata-rata suhu mingguan?
4. Jika suhu nyaman adalah antara 28–30°C, berapa hari yang berada dalam rentang itu?

[15]:
```
# Langkah 1: Menggunakan NamedTuple untuk menyimpan data Suhu

from typing import NamedTuple, List

class WeatherDay(NamedTuple):
    day: str
    temperature: float
```

[16]:
```
# Langkah 2: Membuat Data

# Data awal
data = {
    "day": ["Senin", "Selasa", "Rabu", "Kamis", "Jumat", "Sabtu", "Minggu"],
    "temperature": [28.5, 30.0, 29.2, 27.8, 31.0, 30.5, 28.0]
}

# Buat list WeatherDay
weekly_weather: List[WeatherDay] = [
```

```
    WeatherDay(day, temp) for day, temp in zip(data["day"], data["temperature"])
]
```

[17]:
```
# Langkah 3: Analisis Data
# Kenaikan dan penurunan suhu
max_increase = float('-inf')
max_decrease = float('-inf')
increase_day = ""
decrease_day = ""

for i in range(1, len(weekly_weather)):
    change = weekly_weather[i].temperature - weekly_weather[i - 1].temperature
    if change > max_increase:
        max_increase = change
        increase_day = weekly_weather[i].day
    if change < -max_decrease:  # Note: negative change
        max_decrease = -change
        decrease_day = weekly_weather[i].day

# Rata-rata
avg_temp = sum(w.temperature for w in weekly_weather) / len(weekly_weather)

# Hari nyaman (28 - 30°C)
comfortable_days = [w.day for w in weekly_weather if 28.0 <= w.temperature <=␣
 ↪30.0]
```

[19]:
```
print(f"Hari dengan kenaikan suhu tertinggi: {increase_day} (+{max_increase:.
 ↪1f}°C)")
print(f"Hari dengan penurunan suhu terbesar: {decrease_day} (-{max_decrease:.
 ↪1f}°C)")
print(f"Rata-rata suhu mingguan: {avg_temp:.2f}°C")
print(f"Jumlah hari nyaman (28-30°C): {len(comfortable_days)} hari ({', '.
 ↪join(comfortable_days)})")
```

```
Hari dengan kenaikan suhu tertinggi: Jumat (+3.2°C)
Hari dengan penurunan suhu terbesar: Minggu (-2.5°C)
Rata-rata suhu mingguan: 29.29°C
Jumlah hari nyaman (28-30°C): 4 hari (Senin, Selasa, Rabu, Minggu)
```

1.5 Pustaka Pyrsistent

pyrsistent adalah pustaka Python yang menyediakan struktur data immutable (tidak bisa diubah setelah dibuat), seperti: 1. PVector – seperti list 2. PMap – seperti dict 3. PRecord – seperti NamedTuple / dataclass, tapi immutable dan bisa dengan validasi 4. PSet, PBag – untuk set dan multiset …dan lainnya.

Struktur ini tidak diubah secara in-place, melainkan menghasilkan salinan baru ketika dimodifikasi, cocok untuk: - Pemrograman fungsional - Concurrent/multithreaded environments - Undo/Redo,

audit trail, dan pengolahan data historis

Cara Install:

```
pip install pyrsistent
```

Contoh Dasar (PRecord)

PRecord adalah versi NamedTuple/@dataclass versi immutable dari pyrsistent.

[ ]:
```python
from pyrsistent import PRecord, field

class Point(PRecord):
    x = field(type=float)
    y = field(type=float)

p1 = Point(x=1.5, y=2.0)
print(p1)  # Point(x=1.5, y=2.0)

# p1.x = 100  # Error: Cannot modify (immutable)
p2 = p1.set(x=3.0)  # Mengembalikan salinan baru dengan x diubah
print(p2)  # Point(x=3.0, y=2.0)
# p1 tetap tidak berubah
```

1.5.1 Studi Kasus: Representasi Immutable Graf Jalan Kota

Kita akan membuat representasi graf sederhana untuk jalan kota, di mana: - Node adalah lokasi (Place) - Edge adalah jalan antar lokasi (Road) - Graf menyimpan daftar node dan daftar edge

Tujuan: - Menampilkan daftar jalan dari suatu lokasi - Serialize data ke bentuk dict (untuk JSON, log, dll) - Menunjukkan keuntungan pyrsistent untuk graf fungsional

[1]:
```python
from pyrsistent import PRecord, PVector, PMap, field, pvector

class Place(PRecord):
    name = field(type=str)

    def serialize(self):
        return {"name": self.name}

class Road(PRecord):
    from_place = field(type=Place)
    to_place = field(type=Place)
    distance = field(type=float)

    def serialize(self):
        return {
            "from": self.from_place.name,
            "to": self.to_place.name,
            "distance": self.distance
```

```
        }

class CityGraph(PRecord):
    places = field(type=PVector)        # List of Place
    roads = field(type=PVector)         # List of Road

    def add_place(self, name: str):
        new_place = Place(name=name)
        return self.set(places=self.places.append(new_place))

    def add_road(self, from_name: str, to_name: str, distance: float):
        from_place = next(p for p in self.places if p.name == from_name)
        to_place = next(p for p in self.places if p.name == to_name)
        new_road = Road(from_place=from_place, to_place=to_place,␣
 ↪distance=distance)
        return self.set(roads=self.roads.append(new_road))

    def roads_from(self, place_name: str):
        return [r for r in self.roads if r.from_place.name == place_name]

    def serialize(self):
        return {
            "places": [p.serialize() for p in self.places],
            "roads": [r.serialize() for r in self.roads]
        }
```

[2]:
```
# Inisialisasi graf kosong
city = CityGraph(places=pvector(), roads=pvector())

# Tambah tempat
city = city.add_place("A")
city = city.add_place("B")
city = city.add_place("C")

# Tambah jalan
city = city.add_road("A", "B", 5.2)
city = city.add_road("A", "C", 3.8)
city = city.add_road("B", "C", 2.1)

# Tampilkan jalan dari tempat A
roads_from_a = city.roads_from("A")
for r in roads_from_a:
    print(f"{r.from_place.name} -> {r.to_place.name} ({r.distance} km)")

# Serialisasi (untuk JSON/logging)
import json
print(json.dumps(city.serialize(), indent=2))
```

```
A -> B (5.2 km)
A -> C (3.8 km)
{
  "places": [
    {
      "name": "A"
    },
    {
      "name": "B"
    },
    {
      "name": "C"
    }
  ],
  "roads": [
    {
      "from": "A",
      "to": "B",
      "distance": 5.2
    },
    {
      "from": "A",
      "to": "C",
      "distance": 3.8
    },
    {
      "from": "B",
      "to": "C",
      "distance": 2.1
    }
  ]
}
```