

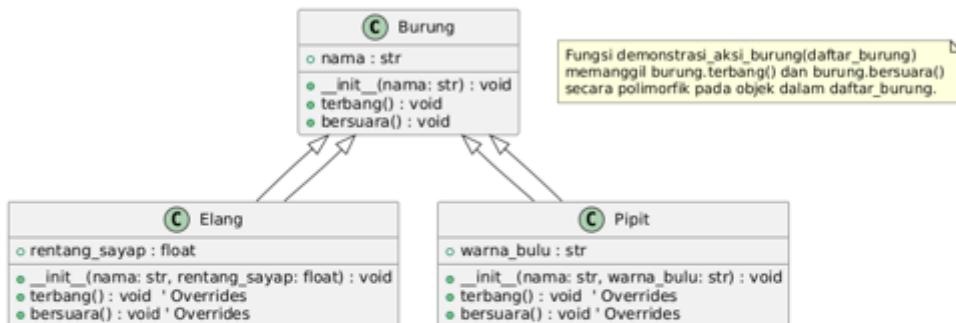


JOBSHEET 05:

POLIMORFISME DAN MEMAHAMI OVERLOADING/OVERRIDING

Praktikum 01: Polimorfisme dengan Inheritance dan Overriding

Tujuan: Mengamati bagaimana objek dari kelas anak yang berbeda merespons pemanggilan metode yang sama (yang di-override) dengan cara yang berbeda, menunjukkan konsep polimorfisme.



Penjelasan Kelas Diagram:

- Kelas Burung:
 - Atribut Publik (+): nama.
 - Metode Publik (+): __init__(nama), terbang() (versi umum), bersuara() (versi umum).
- Kelas Elang (Mewarisi dari Burung):
 - Atribut Publik (+): rentang_sayap (Atribut nama diwarisi).
 - Metode Publik (+): __init__(nama, rentang_sayap) (memanggil super), terbang() (override), bersuara() (override).
- Kelas Pipit (Mewarisi dari Burung):
 - Atribut Publik (+): warna_bulu (Atribut nama diwarisi).
 - Metode Publik (+): __init__(nama, warna_bulu) (memanggil super), terbang() (override), bersuara() (override).

Dosen:

Ir. Prayitno, S.ST., M.T., Ph.D.

Nama Mahasiswa : _____
NIM Mahasiswa : _____



Tahun Akademik 2025

A. Tujuan Instruksional Khusus

Setelah menyelesaikan praktikum ini, mahasiswa diharapkan mampu:

1. Menjelaskan konsep polimorfisme (polymorphism) dalam Pemrograman Berorientasi Objek.
2. Mengimplementasikan polimorfisme melalui pewarisan (inheritance) dan penggantian metode (method overriding).
3. Memahami konsep "duck typing" sebagai bentuk polimorfisme dalam Python.
4. Membedakan antara method overriding (yang didukung Python) dan method overloading (yang tidak didukung secara langsung seperti di bahasa lain).
5. Mengimplementasikan perilaku yang menyerupai overloading menggunakan default arguments, *args, dan **kwargs di Python.

B. Alat dan Bahan

Perangkat Lunak:

- Sistem operasi Windows/Linux/macOS (sesuai ketersediaan).
- Python 3.x (versi terbaru yang stabil).
- IDE atau Code Editor (misalnya, PyCharm, VS Code, atau IDLE).

Perangkat Keras:

- Komputer/Laptop dengan spesifikasi memadai untuk menjalankan Python.

Bahan Pendukung:

- Modul/slide perkuliahan yang menjelaskan dasar pemrograman Python.
- Koneksi internet (opsional, untuk referensi tambahan).

C. Dasar Teori

1. Polimorfisme (Polymorphism)

Polimorfisme berasal dari bahasa Yunani yang berarti "banyak bentuk" (poly = banyak, morph = bentuk). Dalam konteks OOP, polimorfisme merujuk pada kemampuan objek dari kelas yang berbeda untuk merespons terhadap pemanggilan metode yang sama (dengan nama yang sama) dengan cara yang spesifik sesuai dengan tipe atau kelas objek tersebut. Ini memungkinkan kita menulis kode yang lebih generik dan fleksibel, yang dapat beroperasi pada berbagai jenis objek tanpa perlu mengetahui tipe persisnya di muka, asalkan objek-objek tersebut mendukung antarmuka (metode) yang diharapkan.

- **Polimorfisme via Inheritance dan Overriding:** Cara paling umum untuk mencapai polimorfisme dalam bahasa berbasis kelas seperti Python adalah melalui pewarisan. Kelas induk mendefinisikan metode umum, dan kelas anak meng-override metode tersebut untuk menyediakan implementasi spesifik. Kode yang memanggil metode ini pada objek induk dapat secara otomatis menjalankan versi metode yang sesuai dari objek anak yang sebenarnya direferensikan. Contoh: Fungsi yang memanggil metode `terbang()` pada objek Burung akan menjalankan implementasi `terbang()` yang spesifik untuk Elang atau Pipit jika objek tersebut adalah instance dari kelas anak tersebut.
- **Polimorfisme via Duck Typing:** Python adalah bahasa yang diketik secara dinamis (*dynamically typed*), yang seringkali mengandalkan konsep "Duck Typing". Istilah ini berasal dari ungkapan "Jika berjalan seperti bebek dan bersuara seperti bebek, maka itu mungkin bebek." Artinya, Python tidak terlalu peduli dengan *tipe* objek secara eksplisit, tetapi lebih pada apakah objek tersebut memiliki *metode atau atribut yang diperlukan* untuk melakukan suatu operasi. Sebuah fungsi dapat bekerja dengan objek apa pun yang menyediakan metode yang dibutuhkan, terlepas dari kelas atau hierarki pewarisannya.

2. Method Overriding vs. Method Overloading

- **Method Overriding:** Seperti yang dibahas dalam materi Inheritance, overriding terjadi ketika kelas anak menyediakan implementasi ulang (definisi baru) untuk metode yang sudah ada di kelas induknya dengan nama dan signatur (parameter) yang sama. Ini adalah mekanisme kunci untuk polimorfisme dalam hierarki pewarisan. Python sepenuhnya mendukung method overriding.
- **Method Overloading:** Overloading merujuk pada kemampuan untuk mendefinisikan beberapa fungsi atau metode dengan *nama yang sama* dalam lingkup yang sama, tetapi dengan *daftar parameter yang berbeda* (berbeda dalam jumlah atau tipe parameter). Bahasa seperti Java atau C++ menggunakan ini untuk memungkinkan, misalnya, fungsi tambah(int a, int b) dan tambah(float a, float b) ada bersamaan. **Python tidak mendukung method overloading tradisional seperti ini.** Jika Anda mendefinisikan dua fungsi atau metode dengan nama yang sama, definisi terakhir akan menimpas (overwrite) definisi sebelumnya.

3. Menyimulasikan Overloading di Python

Meskipun tidak ada overloading bawaan berdasarkan signatur, Python menyediakan cara lain untuk mencapai fleksibilitas serupa dalam menerima argumen:

- **Nilai Parameter Default:** Memberikan nilai default pada parameter memungkinkan fungsi dipanggil dengan jumlah argumen yang lebih sedikit.

```
Python
def sapa(nama, pesan="Halo"):
    print(f"{pesan}, {nama}!")
sapa("Andi") # Output: Halo, Andi!
sapa("Budi", "Selamat Pagi") # Output: Selamat Pagi, Budi!
```

- **Argumen Posisi Variabel (*args):** Mengizinkan fungsi menerima sejumlah argumen posisi yang tidak ditentukan. Argumen-argumen ini dikumpulkan ke dalam sebuah *tuple*.

```
Python
def jumlahkan(*angka):
    total = 0
    for n in angka:
        total += n
    return total
print(jumlahkan(1, 2, 3)) # Output: 6
print(jumlahkan(5, 10)) # Output: 15
```

- **Argumen Kata Kunci Variabel (**kwargs):** Mengizinkan fungsi menerima sejumlah argumen kata kunci (keyword arguments) yang tidak ditentukan. Argumen-argumen ini dikumpulkan ke dalam sebuah *dictionary*.

```
Python
def cetak_info(**data):
    for kunci, nilai in data.items():
        print(f"{kunci}: {nilai}")
cetak_info(nama="Citra", usia=25, kota="Semarang")
```

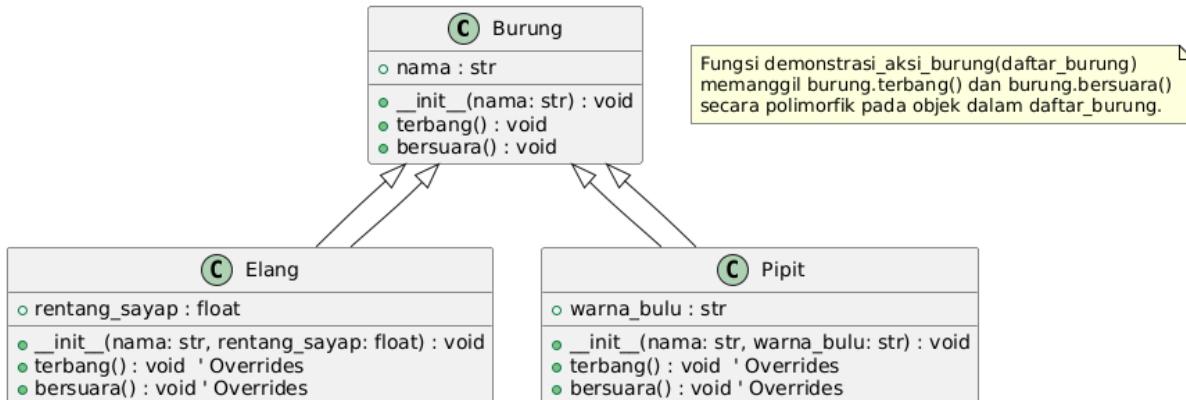
- **Pemeriksaan Tipe/Jumlah di Dalam Fungsi:** Anda dapat memeriksa tipe atau jumlah argumen yang diterima (menggunakan `isinstance()`, `len(args)`, dll.) di dalam fungsi dan menjalankan logika yang berbeda sesuai kondisi.

Dengan menggunakan teknik-teknik ini, kita dapat membuat fungsi dan metode Python yang fleksibel dalam menangani berbagai jenis atau jumlah input, meskipun tidak melalui mekanisme overloading berbasis signatur seperti pada bahasa lain.

D. Langkah Praktikum

Praktikum 01: Polimorfisme dengan Inheritance dan Overriding

Tujuan: Mengamati bagaimana objek dari kelas anak yang berbeda merespons pemanggilan metode yang sama (yang di-override) dengan cara yang berbeda, menunjukkan konsep polimorfisme.



Penjelasan Kelas Diagram:

- **Kelas Burung:**
 - **Atribut Publik (+):** nama.
 - **Metode Publik (+):** __init__(nama), terbang() (versi umum), bersuara() (versi umum).
- **Kelas Elang (Mewarisi dari Burung):**
 - **Atribut Publik (+):** rentang_sayap (Atribut nama diwarisi).
 - **Metode Publik (+):** __init__(nama, rentang_sayap) (memanggil super), terbang() (override), bersuara() (override).
- **Kelas Pipit (Mewarisi dari Burung):**
 - **Atribut Publik (+):** warna_bulu (Atribut nama diwarisi).
 - **Metode Publik (+):** __init__(nama, warna_bulu) (memanggil super), terbang() (override), bersuara() (override).

Diagram ini menunjukkan hierarki kelas dimana Elang dan Pipit mewarisi dari Burung. Kedua kelas anak meng-override metode terbang() dan bersuara() dari kelas induknya. Fungsi demonstrasi_aksi_burung (tidak digambarkan dalam diagram kelas tapi dijelaskan dalam catatan) dapat menerima daftar objek Burung (termasuk instance Elang dan Pipit). Ketika metode terbang() atau bersuara() dipanggil pada elemen dalam daftar tersebut, Python secara otomatis akan menjalankan versi metode yang sesuai dengan tipe objek aktual (versi Elang jika objeknya Elang, versi Pipit jika objeknya Pipit, atau versi Burung jika objeknya Burung), inilah inti dari polimorfisme.

Kemudian untuk kode praktikum dalam python dapat dilihat sebagai berikut:

```

01: # Kelas Induk
02: class Burung:
03:     def __init__(self, nama):
04:         self.nama = nama
05:
06:     def terbang(self):
07:         print(f"{self.nama} terbang dengan cara umum.")
08:
09:     def bersuara(self):
10:         print(f"{self.nama} mengeluarkan suara burung.")
  
```

```

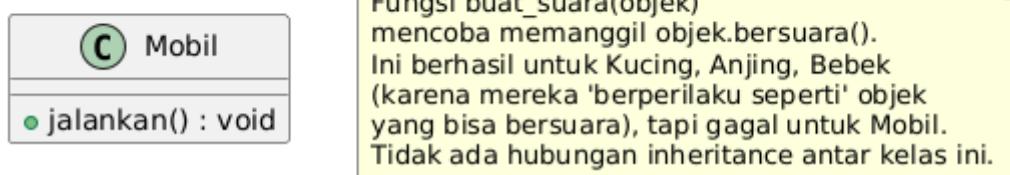
11:
12: # Kelas Anak 1
13: class Elang(Burung):
14:     def __init__(self, nama, rentang_sayap):
15:         super().__init__(nama)
16:         self.rentang_sayap = rentang_sayap
17:
18:     # Override
19:     def terbang(self):
20:         print(f"{self.nama} terbang tinggi melayang di angkasa.")
21:
22:     # Override
23:     def bersuara(self):
24:         print(f"{self.nama} berteriak nyaring!")
25:
26: # Kelas Anak 2
27: class Pipit(Burung):
28:     def __init__(self, nama, warna_bulu):
29:         super().__init__(nama)
30:         self.warna_bulu = warna_bulu
31:
32:     # Override
33:     def terbang(self):
34:         print(f"{self.nama} terbang cepat di antara pepohonan.")
35:
36:     # Override
37:     def bersuara(self):
38:         print(f"{self.nama} berkicau merdu: Cít cít!")
39:
40: # Fungsi yang memanfaatkan polimorfisme
41: def demonstrasi_aksi_burung(daftar_burung):
42:     print("\nAksi Burung:")
43:     for burung in daftar_burung:
44:         print(f"-- Aksi untuk {burung.nama} --")
45:         # Panggilan metode yang sama, tapi perilaku berbeda tergantung
objek
46:         burung.terbang()
47:         burung.bersuara()
48:         print("-" * 15)
49:
50: # --- Kode Utama ---
51: if __name__ == "__main__":
52:     elang_jawa = Elang("Elang Jawa", 1.5)
53:     pipit_gereja = Pipit("Pipit Gereja", "Coklat")
54:     burung_aneh = Burung("Burung Misterius") # Objek dari kelas induk
55:
56:     koleksi_burung = [elang_jawa, pipit_gereja, burung_aneh]
57:
58:     demonstrasi_aksi_burung(koleksi_burung)

```

Observasi: Perhatikan bagaimana pemanggilan `burung.terbang()` dan `burung.bersuara()` di dalam fungsi `demonstrasi_aksi_burung` menghasilkan output yang berbeda-beda tergantung pada tipe objek aktual (Elang, Pipit, atau Burung) yang sedang diproses dalam loop. Ini adalah contoh polimorfisme melalui overriding metode.

Praktikum 02: Polimorfisme dengan Duck Typing

Tujuan: Menunjukkan bagaimana Python dapat menerapkan polimorfisme pada objek yang tidak terikat oleh hierarki pewarisan yang sama, selama objek tersebut memiliki metode yang dibutuhkan ("duck typing").



Penjelasan Kelas Diagram:

- **Kelas Kucing, Anjing, Bebek:** Masing-masing adalah kelas independen yang kebetulan memiliki metode publik `bersuara()` dengan nama yang sama.
- **Kelas Mobil:** Kelas independen yang memiliki metode `jalankan()` tetapi tidak memiliki metode `bersuara()`.
- **Catatan (Note):** Menjelaskan inti dari praktikum ini. Fungsi eksternal `buat_suara` tidak memeriksa tipe kelas secara eksplisit. Ia hanya mencoba memanggil metode `bersuara()` pada objek apa pun yang diterimanya. Jika objek tersebut "berperilaku seperti" sesuatu yang bisa bersuara (yaitu, memiliki metode `bersuara()`), pemanggilan berhasil. Jika tidak (seperti pada objek Mobil), terjadi error (yang ditangani `try-except` dalam kode).

Diagram ini secara visual menunjukkan bahwa kelas-kelas tersebut tidak terhubung melalui pewarisan. Polimorfisme di sini terjadi bukan karena hierarki kelas, melainkan karena kesamaan *interface* (adanya metode `bersuara()` yang dapat dipanggil), yang merupakan ciri khas dari duck typing di Python.

Kode Python:

```

01: class Kucing:
02:     def bersuara(self):
03:         print("Kucing: Meow!")
04:
05: class Anjing:
06:     def bersuara(self):
07:         print("Anjing: Guk guk!")
08:
09: class Bebek:
10:     def bersuara(self):
11:         print("Bebek: Kwek kwek!")
12:
13: class Mobil: # Kelas ini TIDAK punya metode bersuara()
14:     def jalankan(self):
15:         print("Mobil: Brummm!")
16:
17: # Fungsi ini tidak peduli tipe objeknya,
18: # asal punya metode .bersuara()
19: def buat_suara(objek_yang_bisa_bersuara):

```

```

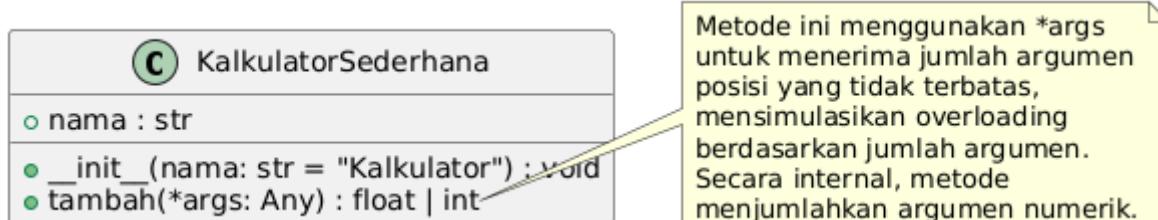
20:     try:
21:         # Memanggil metode 'bersuara' jika ada
22:         objek_yang_bisa_bersuara.bersuara()
23:     except AttributeError:
24:         # Menangani jika objek tidak punya metode 'bersuara'
25:         print(f"Objek {type(objek_yang_bisa_bersuara).__name__} tidak
bisa bersuara.")
26:
27: # --- Kode Utama ---
28: if __name__ == "__main__":
29:     kucing1 = Kucing()
30:     anjing1 = Anjing()
31:     bebek1 = Bebek()
32:     mobil1 = Mobil()
33:
34:     daftar_objek = [kucing1, anjing1, bebek1, mobil1]
35:
36:     print("Demonstrasi Duck Typing:")
37:     for item in daftar_objek:
38:         buat_suara(item) # Fungsi polimorfik via duck typing

```

Observasi: Fungsi `buat_suara` berhasil memanggil metode `bersuara()` pada objek `Kucing`, `Anjing`, dan `Bebek` meskipun mereka tidak mewarisi dari kelas induk yang sama. Fungsi ini hanya peduli apakah objek *memiliki* metode `bersuara()`. Ketika objek `Mobil` (yang tidak memiliki metode `bersuara()`) dimasukkan, fungsi menangani `AttributeError` dengan baik. Ini menunjukkan fleksibilitas duck typing.

Praktikum 03: Simulasi Overloading dalam Metode Kelas dengan *args

Tujuan: Mendemonstrasikan bagaimana sebuah metode di dalam kelas dapat dirancang untuk menerima jumlah argumen yang bervariasi menggunakan `*args`, sehingga meniru perilaku overloading.



Penjelasan Kelas Diagram:

- **Kelas KalkulatorSederhana:**
 - **Atribut Publik (+):** `nama` (string)
 - **Metode Publik (+):**
 - `__init__(nama: str = "Kalkulator") : void`: Konstruktor dengan parameter opsional untuk nama kalkulator.
 - `tambah(*args: Any) : float | int`: Metode tambah yang menerima sejumlah argumen (`*args`). Tipe `Any` menunjukkan fleksibilitas tipe input, dan tipe return bisa `int` atau `float` tergantung hasil penjumlahan.
- **Catatan (Note):** Menjelaskan peran `*args` dalam metode `tambah` untuk menerima jumlah argumen yang bervariasi, yang merupakan cara Python untuk meniru perilaku overloading berdasarkan jumlah parameter.

Diagram ini menunjukkan kelas `KalkulatorSederhana` dengan metode `tambah` yang dirancang secara fleksibel menggunakan `*args`.

Kode Program

```

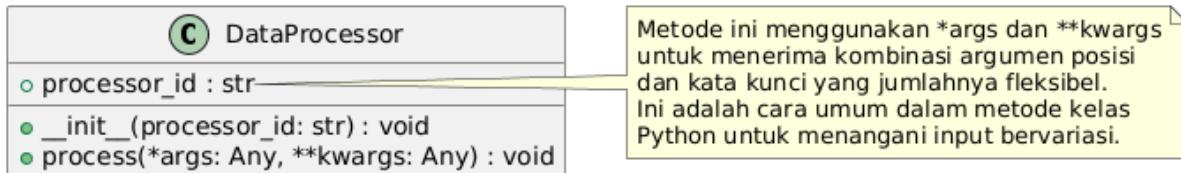
01: class KalkulatorSederhana:
02:     def __init__(self, nama="Kalkulator"):
03:         self.nama = nama
04:         print(f"{self.nama} siap digunakan.")
05:
06:     def tambah(self, *args):
07:         """
08:             Metode ini menjumlahkan semua argumen angka yang diberikan.
09:             Menerima sejumlah argumen posisi menggunakan *args.
10:         """
11:         print(f"\nMemanggil metode tambah dengan argumen: {args}")
12:         if not args:
13:             print("Tidak ada angka untuk dijumlahkan.")
14:             return 0 # Atau bisa juga None atau raise error
15:
16:         total = 0
17:         valid_input = True
18:         for angka in args:
19:             # Melakukan pengecekan tipe sederhana
20:             if isinstance(angka, (int, float)):
21:                 total += angka
22:             else:
23:                 print(f"Peringatan: Argumen '{angka}' bukan angka dan
akan diabaikan.")
24:                 valid_input = False
25:
26:         if valid_input:
27:             print(f"Hasil penjumlahan: {total}")
28:         else:
29:             print(f"Hasil penjumlahan (dengan beberapa input diabaikan):
{total}")
30:         return total
31:
32: # --- Kode Utama ---
33: if __name__ == "__main__":
34:     calc = KalkulatorSederhana("Calc-01")
35:
36:     # Memanggil metode 'tambah' dengan jumlah argumen berbeda
37:     print("\n--- Percobaan Penjumlahan ---")
38:     calc.tambah(5, 10)
39:     calc.tambah(2, 3, 5, 10)
40:     calc.tambah(100)
41:     calc.tambah() # Tanpa argumen
42:     calc.tambah(1, 2, "tiga", 4, 5.5) # Dengan input non-angka

```

Observasi: Perhatikan bagaimana metode `tambah` dari objek `calc` dapat dipanggil dengan dua, empat, satu, atau bahkan nol argumen angka. Penggunaan `*args` mengumpulkan semua argumen posisi ke dalam sebuah tuple `args`, yang kemudian diiterasi di dalam metode untuk melakukan penjumlahan. Metode ini juga menunjukkan penanganan sederhana jika input bukan angka atau jika tidak ada argumen yang diberikan. Ini mensimulasikan kemampuan fungsi/metode untuk bekerja dengan jumlah input yang berbeda.

Praktikum 04: Simulasi Overloading dalam Metode Kelas dengan *args dan **kwargs

Tujuan: Menggunakan *args dan **kwargs di dalam sebuah metode kelas untuk menerima jumlah argumen posisi dan kata kunci yang fleksibel, meniru perilaku metode yang bisa dipanggil dengan cara berbeda.



Penjelasan Kelas Diagram:

- **Kelas DataProcessor:**
 - **Atribut Publik (+):** processor_id (string).
 - **Metode Publik (+):**
 - `__init__(processor_id: str) : void`: Konstruktor untuk inisialisasi ID prosesor.
 - `process(*args: Any, **kwargs: Any) : void`: Metode process yang menerima self (implisit dalam diagram), sejumlah argumen posisi (*args), dan sejumlah argumen kata kunci (**kwargs).
- **Catatan (Note):** Menjelaskan fungsi *args dan **kwargs dalam konteks metode process untuk memberikan fleksibilitas dalam menerima argumen, sebuah teknik yang digunakan untuk tujuan serupa dengan overloading di bahasa lain.

Diagram ini menunjukkan kelas DataProcessor dan metode process yang dirancang untuk menangani berbagai jenis pemanggilan argumen menggunakan *args dan **kwargs.

Kode Program:

```

01: class DataProcessor:
02:     def __init__(self, processor_id):
03:         self.processor_id = processor_id
04:         print(f"Data Processor {self.processor_id} siap.")
05:
06:     def process(self, *args, **kwargs):
07:         """
08:             Metode ini memproses data dengan argumen posisi (*args)
09:             dan kata kunci (**kwargs) yang jumlahnya variabel.
10:         """
11:         print(f"\n--- {self.processor_id} Memproses Data ---")
12:
13:         # Memproses argumen posisi (*args -> tuple)
14:         if args:
15:             print("Argumen Posisi Diterima:")
16:             for i, arg in enumerate(args):
17:                 print(f" args[{i}]: {arg} (tipe: {type(arg).__name__})")
18:         else:
19:             print("Tidak ada argumen posisi.")
20:
21:         # Memproses argumen kata kunci (**kwargs -> dict)
22:         if kwargs:
23:             print("Argumen Kata Kunci Diterima:")
24:             for kunci, nilai in kwargs.items():
25:                 print(f" {kunci}: {nilai} (tipe: {type(nilai).__name__})")
26:         else:
27:             print("Tidak ada argumen kata kunci.")
28:         print("-----")
29:
30: # --- Kode Utama ---
  
```

```

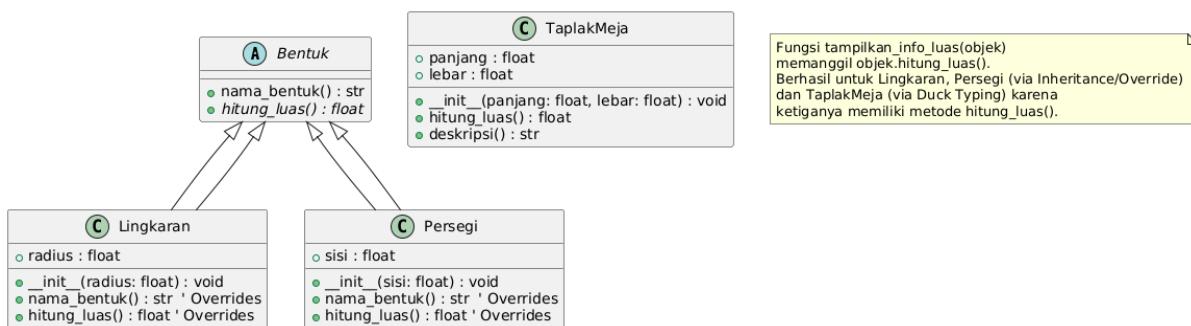
31: if __name__ == "__main__":
32:     # Membuat instance dari kelas
33:     processor1 = DataProcessor("DP-001")
34:
35:     # Memanggil metode 'process' dengan berbagai cara
36:     print("\nPanggilan 1: Tanpa argumen tambahan")
37:     processor1.process()
38:
39:     print("\nPanggilan 2: Hanya argumen posisi")
40:     processor1.process(100, "Status OK", 99.9, False)
41:
42:     print("\nPanggilan 3: Hanya argumen kata kunci")
43:     processor1.process(user="admin", level=5, mode="verbose")
44:
45:     print("\nPanggilan 4: Kombinasi argumen posisi dan kata kunci")
46:         processor1.process("Task-A", "Task-B", status="Running",
priority="High", thread_id=54321)
47:
48:     # Panggilan dengan list dan dictionary sebagai argumen posisi
49:     print("\nPanggilan 5: Argumen posisi kompleks")
50:     list_ids = ["A1", "B2", "C3"]
51:     dict_params = {"timeout": 30, "retry": 3}
52:         processor1.process(list_ids, dict_params, owner="system",
enabled=True)

```

Observasi: Perhatikan bagaimana metode `process` milik objek `processor1` dapat dipanggil dengan berbagai kombinasi argumen. `*args` dan `**kwargs` dalam definisi metode `process(self, *args, **kwargs)` memungkinkan fleksibilitas ini. `self` tetap menjadi parameter pertama yang merujuk pada instance objek, diikuti oleh `*args` untuk argumen posisi dan `**kwargs` untuk argumen kata kunci.

Praktikum 05: Kombinasi Polimorfisme (Inheritance & Duck Typing)

Tujuan: Menunjukkan bagaimana sebuah fungsi dapat bekerja secara polimorfik dengan objek-objek dari hierarki inheritance yang sama *dan* objek dari kelas lain yang tidak terkait, selama semuanya menyediakan metode yang dibutuhkan.



Penjelasan Kelas Diagram:

- **Kelas Bentuk (Abstrak):**
 - Kelas induk yang mendefinisikan interface umum. Metode `hitung_luas()` ditandai `{abstract}` untuk menunjukkan bahwa ia harus diimplementasikan oleh subclass.
- **Kelas Lingkaran dan Persegi:**
 - Mewarisi dari Bentuk.
 - Masing-masing memiliki atribut spesifik (`radius`, `sisi`).

- Meng-override metode `nama_bentuk()` dan `hitung_luas()` dari kelas induk.

- **Kelas TaplakMeja:**

- Kelas yang **tidak** mewarisi dari Bentuk.
- Memiliki atribut panjang, lebar.
- Memiliki metode `hitung_luas()` yang **kebetulan namanya sama** dengan metode di Bentuk.
- Memiliki metode lain (deskripsi) yang tidak ada di Bentuk.

Catatan (Note): Menjelaskan bagaimana fungsi `tampilkan_info_luas` berinteraksi dengan objek-objek ini. Ia bekerja pada Lingkaran dan Persegi karena mereka adalah turunan Bentuk (polimorfisme via inheritance). Ia juga bekerja pada TaplakMeja karena TaplakMeja memiliki metode `hitung_luas()` yang dibutuhkan (polimorfisme via duck typing), meskipun tidak ada hubungan inheritance formal.

Kode Program:

```

01: import math
02:
03: # --- Definisi Kelas dari Hierarki Inheritance ---
04: class Bentuk:
05:     # Metode ini bisa dianggap sebagai 'template' atau interface dasar
06:     def nama_bentuk(self):
07:         return "Bentuk Generik"
08:
09:     # Metode ini wajib di-override oleh subclass
10:     def hitung_luas(self):
11:         raise NotImplementedError("Subclass harus mengimplementasikan
metode ini")
12:
13: class Lingkaran(Bentuk):
14:     def __init__(self, radius):
15:         self.radius = radius
16:
17:     # Override
18:     def nama_bentuk(self):
19:         return "Lingkaran"
20:
21:     # Override
22:     def hitung_luas(self):
23:         return math.pi * (self.radius ** 2)
24:
25: class Persegi(Bentuk):
26:     def __init__(self, sisi):
27:         self.sisi = sisi
28:
29:     # Override
30:     def nama_bentuk(self):
31:         return "Persegi"
32:
33:     # Override
34:     def hitung_luas(self):
35:         return self.sisi * self.sisi
36:
37: # --- Definisi Kelas Lain (Tidak Terkait Inheritance dengan Bentuk) ---
38: class TaplakMeja:
39:     def __init__(self, panjang, lebar):
40:         self.panjang = panjang
41:         self.lebar = lebar
42:
43:     # Metode dengan nama sama -> 'hitung_luas'
```

```

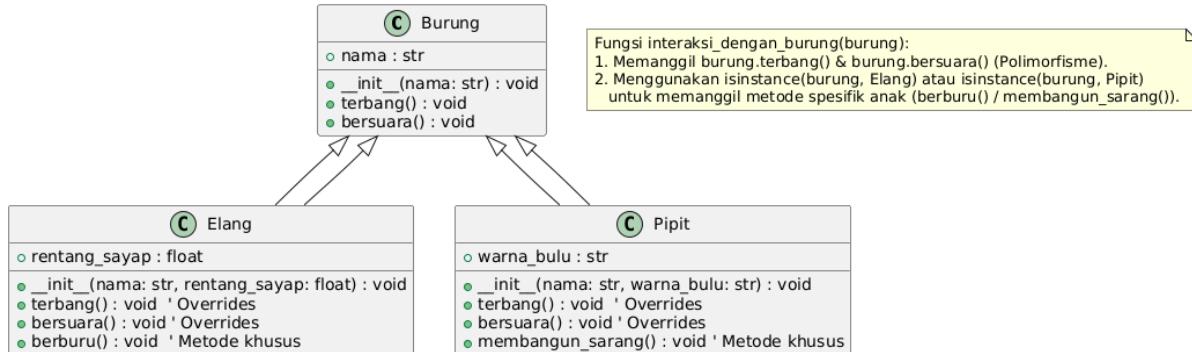
44:     # Ini memungkinkan TaplakMeja bekerja dengan fungsi polimorfik di
bawah
45:     def hitung_luas(self):
46:         return self.panjang * self.lebar
47:
48:     # Metode yang berbeda, tidak ada di kelas Bentuk
49:     def deskripsi(self):
50:         return f"Taplak Meja {self.panjang}x{self.lebar}"
51:
52: # --- Fungsi Polimorfik ---
53: def tampilan_info_luas(objek_dengan_luas):
54:     print("-" * 20)
55:     try:
56:         # Mencoba memanggil hitung_luas() - inti polimorfisme
57:         luas = objek_dengan_luas.hitung_luas()
58:
59:         # Mencoba mendapatkan nama jika ada (duck typing tambahan)
60:         try:
61:             nama = objek_dengan_luas.nama_bentuk()
62:         except AttributeError:
63:             # Jika tidak ada nama_bentuk(), gunakan nama kelasnya
64:             nama = type(objek_dengan_luas).__name__
65:
66:         print(f"Objek: {nama}")
67:         print(f"Luas : {luas:.2f}") # Format luas 2 angka desimal
68:
69:     except AttributeError:
70:         # Menangani jika objek sama sekali tidak punya .hitung_luas()
71:         print(f"Objek {type(objek_dengan_luas).__name__} tidak dapat
dihitung luasnya (metode tidak ditemukan).")
72:     except NotImplementedError:
73:         # Menangani jika metode ada tapi belum diimplementasi (dari kelas
Bentuk)
74:             print(f"Metode 'hitung_luas' belum diimplementasi untuk
{type(objek_dengan_luas).__name__}.")
75:
76:
77: # --- Kode Utama ---
78: if __name__ == "__main__":
79:     lingkaral1 = Lingkaran(7)
80:     persegil1 = Persegi(5)
81:     taplak1 = TaplakMeja(1.5, 0.8)
82:     bentuk_dasar = Bentuk() # Objek kelas induk (akan raise
NotImplementedError)
83:     string_biasa = "Ini string" # Objek yang tidak relevan
84:
85:     daftar_item = [lingkaral1, persegil1, taplak1, bentuk_dasar,
string_biasa]
86:
87:     print("Menampilkan Info Luas (Polimorfisme Campuran):")
88:     for item in daftar_item:
89:         tampilan_info_luas(item)

```

Observasi: Fungsi tampilan_info_luas bekerja dengan Lingkaran dan Persegi karena mereka mewarisi dari Bentuk dan mengimplementasikan hitung_luas(). Fungsi ini juga bekerja dengan TaplakMeja karena, meskipun tidak terkait inheritance, ia memiliki metode hitung_luas() (duck typing). Fungsi ini juga dapat menangani objek Bentuk (yang metodenya belum diimplementasi) dan objek string_biasa (yang tidak memiliki metode yang relevan) dengan try-except.

Praktikum 06: Kontrol Perilaku Polimorfik dengan `isinstance()`

Tujuan: Menggunakan `isinstance()` di dalam fungsi yang bekerja secara polimorfik untuk menambahkan perilaku khusus berdasarkan tipe objek, selain memanggil metode yang di-override.



Penjelasan Kelas Diagram:

- **Kelas Burung, Elang, Pipit:** Struktur hierarki sama seperti Praktikum 1, namun kelas Elang dan Pipit sekarang memiliki metode tambahan yang spesifik untuk mereka (`berburu()` dan `membangun_sarang()`).
- **Catatan (Note):** Menjelaskan bagaimana fungsi `interaksi_dengan_burung` (yang berada di luar kelas) menggunakan dua pendekatan:
 1. Memanfaatkan polimorfisme dengan memanggil metode `terbang()` dan `bersuara()` yang perilakunya bergantung pada tipe objek aktual.
 2. Menggunakan `isinstance()` untuk memeriksa tipe spesifik objek (Elang atau Pipit) agar dapat memanggil metode unik yang hanya dimiliki oleh subclass tersebut.

Diagram ini menunjukkan struktur kelas yang memungkinkan polimorfisme, dan catatan menjelaskan bagaimana logika eksternal dapat menggunakan `isinstance()` untuk berinteraksi secara lebih spesifik dengan objek dalam hierarki tersebut.

Kode Program:

```

01: # --- Definisi Kelas Burung, Elang, Pipit ---
02: class Burung:
03:     def __init__(self, nama):
04:         self.nama = nama
05:     def terbang(self):
06:         print(f"{self.nama} terbang dengan cara umum.")
07:     def bersuara(self):
08:         print(f"{self.nama} mengeluarkan suara burung.")
09:
10: class Elang(Burung):
11:     def __init__(self, nama, rentang_sayap):
12:         super().__init__(nama)
13:         self.rentang_sayap = rentang_sayap
14:     # Override
15:     def terbang(self):
16:         print(f"{self.nama} terbang tinggi melayang di angkasa.")
17:     # Override
18:     def bersuara(self):
19:         print(f"{self.nama} berteriak nyaring!")
20:     # Metode khusus Elang
21:     def berburu(self):
22:         print(f"{self.nama} sedang mencari mangsa dari ketinggian.")
23:
24: class Pipit(Burung):
  
```

```

25:     def __init__(self, nama, warna_bulu):
26:         super().__init__(nama)
27:         self.warna_bulu = warna_bulu
28:     # Override
29:     def terbang(self):
30:         print(f"{self.nama} terbang cepat di antara pepohonan.")
31:     # Override
32:     def bersuara(self):
33:         print(f"{self.nama} berkicau merdu: Cit cit!")
34:     # Metode khusus Pipit
35:     def membangun_sarang(self):
36:         print(f"{self.nama} sedang mengumpulkan ranting untuk sarang.")
37:
38: # --- Fungsi yang memanfaatkan polimorfisme DAN isinstance() ---
39: def interaksi_dengan_burung(burung):
40:     print(f"\n--- Berinteraksi dengan {type(burung).__name__}:
{getattr(burung, 'nama', 'Objek tidak dikenal')} ---")
41:
42:     # Cek dulu apakah objek adalah instance dari Burung atau turunannya
43:     if isinstance(burung, Burung):
44:         # Perilaku polimorfik dasar (memanggil metode override)
45:         burung.terbang()
46:         burung.bersuara()
47:
48:         # Menambahkan perilaku spesifik berdasarkan tipe turunan
49:         if isinstance(burung, Elang):
50:             print("-> Ini adalah Elang!")
51:             burung.berburu() # Panggil metode khusus Elang
52:         elif isinstance(burung, Pipit):
53:             print("-> Ini adalah Pipit!")
54:             burung.membangun_sarang() # Panggil metode khusus Pipit
55:         else:
56:             # Hanya Burung generik, bukan turunan spesifik yang dikenali
57:             print("-> Ini adalah burung jenis umum (bukan Elang/Pipit).")
58:     else:
59:         # Jika objek BUKAN instance Burung sama sekali
60:         print("-> Objek ini bukan termasuk jenis Burung.")
61:     print("-" * 25)
62:
63: # --- Kode Utama ---
64: if __name__ == "__main__":
65:     elang_sumatra = Elang("Elang Sumatra", 1.8)
66:     pipit_rumah = Pipit("Pipit Rumah", "Abu-abu")
67:     merak = Burung("Merak") # Contoh Burung lain (induk)
68:     kucing_tetangga = "Meong" # Objek non-burung
69:
70:     koleksi_makhluk = [elang_sumatra, pipit_rumah, merak,
kucing_tetangga]
71:
72:     for makhluk in koleksi_makhluk:
73:         interaksi_dengan_burung(makhluk)

```

Observasi: Fungsi `interaksi_dengan_burung` pertama-tama memeriksa apakah objek merupakan Burung. Jika ya, ia memanggil metode `terbang()` dan `bersuara()` secara polimorfik. Kemudian, ia menggunakan `isinstance()` lagi untuk pemeriksaan yang lebih spesifik (Elang atau Pipit) dan memicu perilaku tambahan (`berburu()` atau `membangun_sarang()`) yang hanya dimiliki oleh kelas anak tersebut. Jika objek bukan Burung, pesan berbeda ditampilkan. Ini menunjukkan cara menggabungkan polimorfisme dinamis (memanggil metode yang mungkin di-override) dengan pemeriksaan tipe eksplisit untuk kontrol alur program yang lebih spesifik.

E. Hasil Praktikum

Lengkapi hasil tabel praktikum berikut:

No.	Nama Praktikum	Hasil Praktikum
1	Praktikum 1: Polimorfisme dengan Inheritance dan Overriding	
2	Praktikum 2: Polimorfisme dengan Duck Typing	
3	Praktikum 3: Simulasi Overloading dalam Metode Kelas dengan *args	
4	Praktikum 4: Simulasi Overloading dalam Metode Kelas dengan *args dan **kwargs	
5	Praktikum 5: Kombinasi Polimorfisme (Inheritance & Duck Typing)	
6	Praktikum 6: Kontrol Perilaku Polimorfik dengan isinstance()	

F. Penugasan

1. Kumpulkan Laporan Praktikum dari jobsheet ini dalam bentuk Microsoft word sesuai dengan format jobsheet praktikum dan dikumpulkan di web LMS. (JANGAN DALAM BENTUK PDF)
 2. Kumpulkan luaran kode praktikum dalam bentuk ipynb yang sudah diunggah pada akun github masing-masing. Lampirkan tautan github yang sudah diunggah melalui laman LMS.
 3. **Buat Program dan Kelas diagram** untuk hierarki kelas Komputer dan mendemonstrasikan polimorfisme serta simulasi overloading.
- **Buat Kelas Induk (Komputer):**
 - Atribut: merk (string), processor (string), ram_gb (integer).
 - Konstruktor (`__init__`): Menerima dan menginisialisasi ketiga atribut di atas.
 - Metode `info_spesifikasi(self)`: Mencetak informasi dasar komputer (merk, processor, RAM). Contoh:
 - Merk: [Merk Komputer]
 - Processor: [Nama Processor]
 - RAM: [Jumlah RAM] GB
 - Metode `jalankan_aplikasi(self, nama_aplikasi)`: Mencetak pesan bahwa komputer sedang menjalankan aplikasi. Contoh: "[Merk Komputer] menjalankan aplikasi: [nama_aplikasi]..."
 - **Buat Kelas Anak 1 (Laptop):**
 - Mewarisi dari Komputer.
 - Atribut Tambahan: ukuran_layar_inch (float), berat_kg (float).
 - Konstruktor (`__init__`): Menerima merk, processor, ram_gb, ukuran_layar_inch, dan berat_kg. Gunakan `super().__init__()` untuk inisialisasi bagian Komputer.
 - **Override** metode `info_spesifikasi(self)`: Panggil `super().info_spesifikasi()`, lalu tambahkan informasi ukuran layar dan berat laptop. Contoh output tambahan:
 - Ukuran Layar: [Ukuran Layar] inch
 - Berat: [Berat Laptop] Kg
 - **Buat Kelas Anak 2 (Desktop):**
 - Mewarisi dari Komputer.
 - Atribut Tambahan: jenis_casing (string, misal: "Tower", "Mini-PC"), monitor_external (boolean, True jika ada monitor terpisah).
 - Konstruktor (`__init__`): Menerima merk, processor, ram_gb, jenis_casing, dan monitor_external. Gunakan `super().__init__()` untuk inisialisasi bagian Komputer.

- **Override** metode `info_spesifikasi(self)`: Panggil `super().info_spesifikasi()`, lalu tambahkan informasi jenis casing dan status monitor eksternal. Contoh output tambahan:
- Jenis Casing: [Jenis Casing]
- Monitor External: [Ya / Tidak]
- **Demonstrasi Polimorfisme:**
 - Buat sebuah fungsi terpisah, misalnya `cetak_semua_spesifikasi(daftar_komputer)`.
 - Fungsi ini menerima sebuah list `daftar_komputer` yang berisi objek-objek dari Laptop dan Desktop.
 - Di dalam fungsi, lakukan loop pada `daftar_komputer` dan untuk setiap komputer dalam list, panggil metode `komputer.info_spesifikasi()`.
 - Amati bagaimana pemanggilan metode yang sama menghasilkan output yang berbeda (sesuai override di Laptop dan Desktop).
- **Simulasi Overloading (Pilih salah satu metode berikut):**
 - **Opsi A (Default Argument):** Tambahkan metode `upgrade_ram(self, tambahan_gb)` pada kelas Komputer. Modifikasi metode ini agar bisa menerima argumen opsional `tipe_ram` dengan nilai default (misal, "DDR4"). Metode ini harus mencetak informasi upgrade. Contoh: RAM [Merk Komputer] diupgrade sebesar [tambahan_gb] GB (Tipe: [tipe_ram]). Total RAM sekarang: [RAM baru] GB. Panggil metode ini dengan dan tanpa argumen `tipe_ram`.
 - ***Opsi B (args):** Tambahkan metode `install_software(self, *args)` pada kelas Komputer. Metode ini harus bisa menerima satu atau lebih nama software (string) sebagai argumen posisi. Di dalam metode, cetak nama setiap software yang diinstal. Contoh: Menginstall [nama_software] di [Merk Komputer]... Panggil metode ini dengan satu nama software dan dengan beberapa nama software.
- **Buat Kode Utama (Bagian `if __name__ == "__main__":`)**
 - Buat minimal satu objek Laptop dan satu objek Desktop.
 - Masukkan objek-objek tersebut ke dalam sebuah list.
 - Panggil fungsi `cetak_semua_spesifikasi()` yang Anda buat di langkah 4 dengan list tersebut.
 - Demonstrasikan pemanggilan metode simulasi overloading yang Anda pilih di langkah 5 pada salah satu objek komputer Anda.

G. Kesimpulan

Melalui praktikum ini, mahasiswa telah memahami konsep polimorfisme sebagai kemampuan objek berbeda untuk merespons metode yang sama dengan cara unik, baik melalui mekanisme pewarisan dan method overriding maupun melalui pendekatan fleksibel *duck typing* yang umum di Python. Selain itu, telah dipelajari perbedaan antara method overriding yang didukung penuh dan keterbatasan method overloading tradisional di Python, serta bagaimana menyimulasikan fleksibilitas overloading menggunakan teknik seperti nilai parameter default, `*args`, dan `**kwargs`. Penguasaan konsep-konsep ini memungkinkan penulisan kode yang lebih adaptif, dapat diperluas, dan mampu berinteraksi dengan beragam tipe objek secara elegan dalam paradigma Pemrograman Berorientasi Objek.

H. Daftar Pustaka

1. Lutz, M. (2013). *Learning Python*. O'Reilly Media.
2. Guttag, J. V. (2016). *Introduction to Computation and Programming Using Python*. MIT Press.
3. Python Software Foundation. *Python 3 Documentation*. <https://docs.python.org/3/>