Python: Pengenalan dan Konsep OOP

Dalam proses penulisan program, kita biasanya akan terlibat sebuah “diskusi” tentang kapan struktur sebuah kode dikatakan bagus. Apakah yang menganut konsep Object Oriented Programming (OOP), atau yang mengikuti konsep prosedural, atau bahkan konsep fungsional?

Mungkin bagi pemula, kita belum pernah menulis kode program yang cukup besar sampai ribuan baris kode dan puluhan bahkan ratusan file.

Kalau kita sudah pernah melakukan hal itu –*entah projek akhir sekolah atau projek akhir kuliah*, kita baru akan sadar bahwa ternyata semua hal bisa menjadi sangat rumit jika kita tidak mengadopsi pendekatan yang baik.

Pada tulisan kali ini –*dan insyaallah beberapa tulisan kedepan*, kita akan membahas tentang Object Oriented Programming (OOP) atau yang biasa dialihbahasakan menjadi Pemrograman Berorientasi Objek (PBO) –*tentunya tetap dalam bingkai bahasa pemrograman Python*.

Pengertian Paradigma Pemrograman

Apa yang dimaksud dengan paradigma pemrograman?

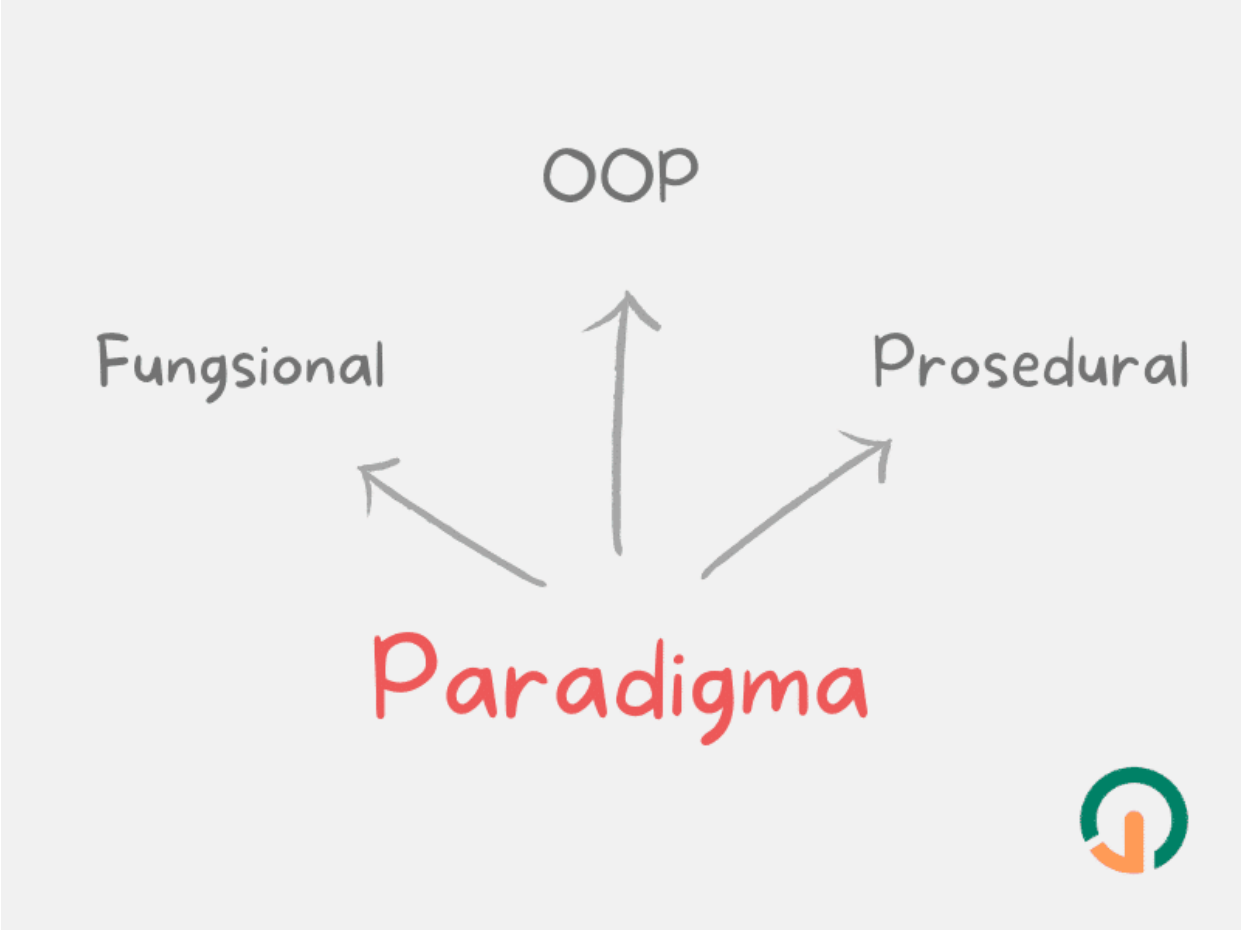
Kata “paradigma” sebenarnya tidak merujuk pada suatu bahasa pemrograman tertentu. Ia lebih merujuk kepada sebuah pendekatan atau konsep dalam menstrukturisasi suatu kode program [1]. Atau, kita bisa juga menyebut paradigma sebagai “cara berfikir” untuk menyusun pola penulisan program.

Sebagian bahasa pemrograman dibuat mudah untuk mengadopsi suatu paradigma tertentu, dan sebagian bahasa yang lain juga dibuat lebih mudah untuk mengadopsi paradigma yang lain.

3 Paradigma Pemrograman yang Cukup Populer

Di antara berbagai macam paradigma pemrograman, terdapat 3 paradigma yang setidaknya cukup populer (bahkan pertanyaan tersebut mendapatkan lebih dari 191k views di stackoverflow [2]).

Ketiga paradigma tersebut adalah:



- Paradigma prosedural – melakukan pengelompokan satu tugas tertentu yang bisa digunakan berkali-kali sebagai pendekatan dalam pemecahan suatu masalah.
- Paradigma fungsional – hampir sama dengan prosedural, hanya saja paradigma ini lebih berorientasi terhadap “input-output” dari pada mengubah data secara langsung seperti pada paradigma prosedural.
- dan Paradigma PBO (Pemrograman Berorientasi Objek) – yaitu paradigma yang menjadikan semua komponen program sebagai objek. Yang mana setiap objek memiliki identitas dan tugasnya masing-masing.

Apa Perbedaan Paradigma Prosedural dan Fungsional?

Untuk prosedural dan fungsional, sebenarnya tidak jauh berbeda. Ketika dulu saya mengambil mata kuliah Algoritma Pemrograman, salah seorang dosen kala itu memang membedakan antara fungsi dan prosedur:

Fungsi adalah fungsi yang mengembalikan nilai.

Sedangkan prosedur adalah fungsi yang tidak mengembalikan nilai.

Ada pun pada pelajaran-pelajaran bahasa pemrograman yang lebih modern, kebanyakan orang tidak lagi membedakan antara fungsi yang mengembalikan nilai dan fungsi yang tidak mengembalikan nilai, baik secara istilah mau pun secara penulisan sintaks (berbeda dengan beberapa bahasa pemrograman low level yang mana sintaks pembuatan fungsi dan prosedur memang dibuat berbeda).

Mari kita lihat perbedaannya secara langsung

Untuk lebih jelasnya, kita coba buat contoh aplikasi yang menggunakan pendekatan prosedural dan juga pendekatan fungsional.

Aplikasi yang kita buat adalah aplikasi untuk menghitung luas segitiga.

1. Pendekatan Prosedural

```
def hitung_luas ():
  alas = float(input('Masukkan alas: '))
  tinggi = float(input('Masukkan tinggi: '))
  print('Luas =', 0.5 * alas * tinggi)

# kita bisa panggil berkali-kali
hitung_luas()
# panggil lagi
hitung_luas()
```

2.

3. Pendekatan Fungsional

```
def input_alas_dan_tinggi ():
  alas = float(input('Masukkan alas: '))
  tinggi = float((input('Masukkan tinggi: ')))

  return alas, tinggi

def hitung_luas (alas, tinggi):
  return 0.5 * alas * tinggi

"""kalau fungsional, kita sendiri yang mengelola
hasil kembaliannya"""

# satu fungsi bisa dipanggil secara independen
print(hitung_luas(5, 10))

# contoh dengan inputan alas dan tinggi
alas, tinggi = input_alas_dan_tinggi()
print(hitung_luas(alas, tinggi))
```

4.

Contoh output dari 2 kode program di atas:

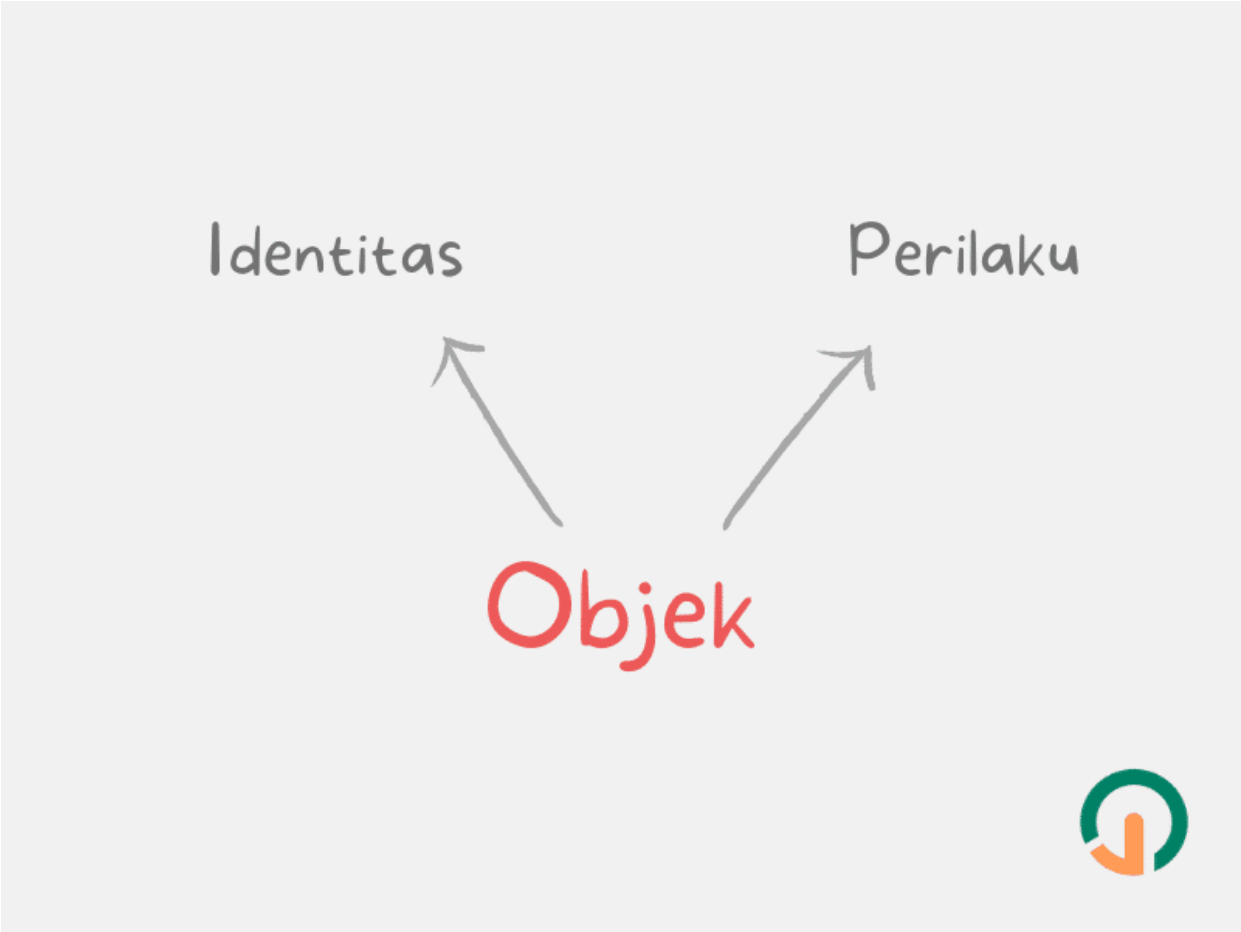
```
Masukkan alas: 5
Masukkan tinggi: 10
Luas = 25.0
Masukkan alas: 20
Masukkan tinggi: 5
Luas = 50.0
```

Konsep Dasar Pemrograman Berorientasi Objek (PBO)

Paradigma prosedural dan fungsional memang sekilas mirip dan memiliki banyak persamaan (baik dari sintaks mau pun dari penggunaan fungsi).

Tapi berbeda dengan konsep paradigma PBO yang dari cara berpikir saja sudah berbeda.

Pada konsep PBO, kita akan membuat semua komponen program seolah-olah adalah sebuah objek. Sebuah objek selalu memiliki identitas dan juga perilaku atau kemampuan untuk melakukan tugas tertentu.

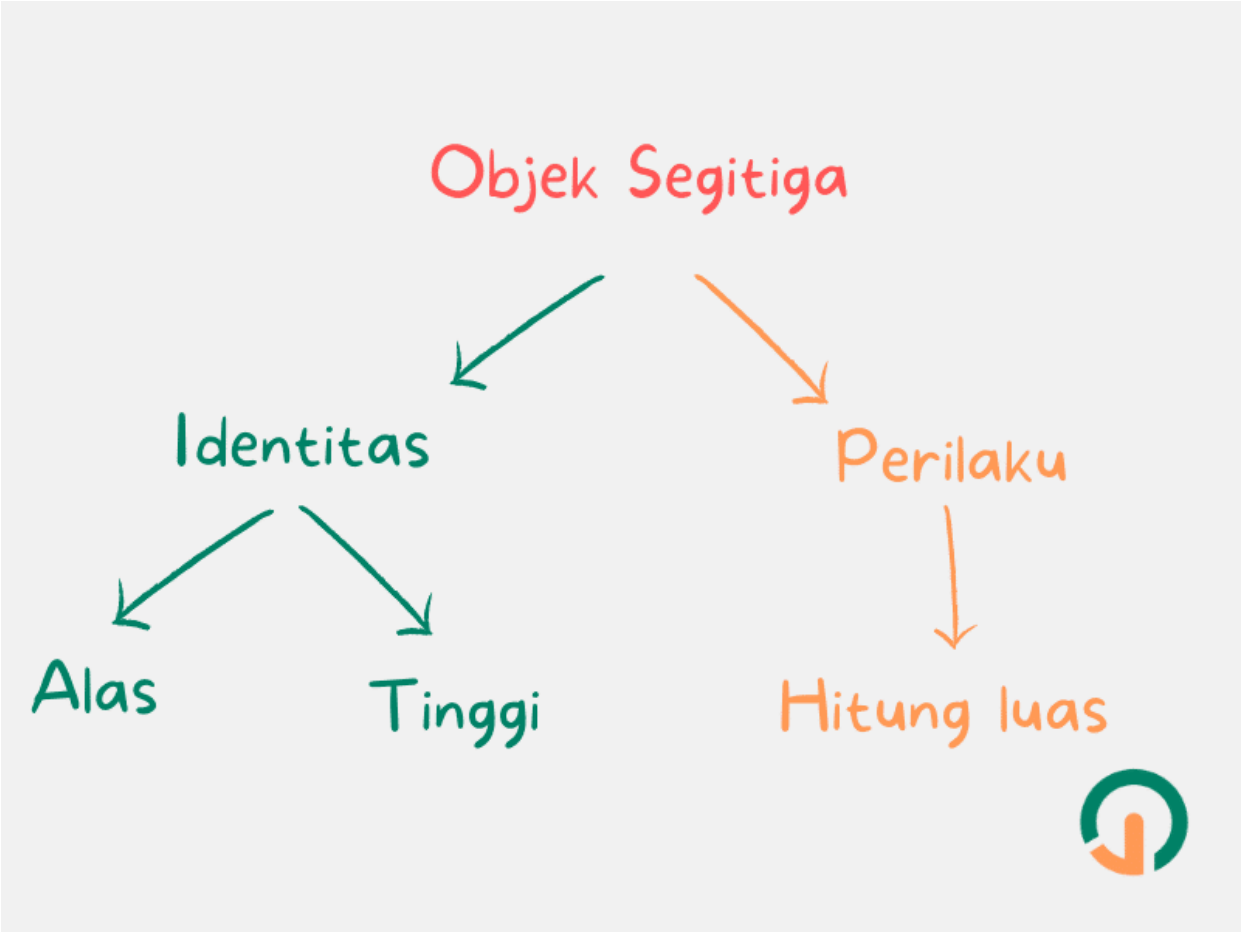


Kita ambil contoh saja program yang ada di atas: cara menghitung segitiga.

Maka cara berpikir kita akan seperti ini:

1. Segitiga adalah sebuah objek.
2. Objek segitiga memiliki 2 identitas berupa `alas` dan `tinggi`.
3. Objek segitiga memiliki “kemampuan” untuk menghitung luasnya sendiri.

Kalau kita ilustrasikan akan terlihat seperti ini:



Pemrograman Beriorientasi Objek Pada Python

Kita insyaallah akan membahas lebih dalam pada pertemuan berikutnya tentang bagaimana menerapkan PBO pada Python.

Akan tetapi sebagai gambaran awal, berikut ini kode program untuk menyelesaikan perhitungan luas segitiga dengan paradigma OOP / PBO:

```
class Segitiga:
  def __init__(self, alas, tinggi):
    self.alas = alas
    self.tinggi = tinggi

  def get_luas(self):
    return 0.5 * self.alas * self.tinggi
```

```
segitiga1 = Segitiga(5, 10)
segitiga2 = Segitiga(10, 10)

print('luas segitiga1:', segitiga1.get_luas())
print('luas segitiga2:', segitiga2.get_luas())
```

Kesimpulan

1. Sebuah paradigma pemrograman sangat penting untuk dipertimbangkan dengan baik ketika kita mulai menulis sebuah program yang besar.
2. Terdapat beberapa paradigma pemrograman, di antara yang paling populer adalah paradigma fungsional, prosedural, dan OOP.
3. Paradigma fungsional dan prosedural sangat bergantung pada sebuah fungsi. Bedanya, paradigma fungsional menggunakan fungsi yang mengembalikan data, sedangkan prosedural menggunakan fungsi yang tidak mengembalikan data (meskipun kita bisa melakukan penggabungan *mix and match*).
4. Paradigma OOP memaksa cara berpikir kita untuk menganggap setiap komponen dari aplikasi sebagai objek.

Python: Kelas Dan Objek

Semua Hal Pada Python Adalah Objek

Sebelumnya, kita perlu tahu dulu bahwa sesuatu tidaklah dikatakan objek kecuali jika memiliki atribut atau perilaku.

Atribut adalah semacam identitas atau variabel dari suatu objek, sedangkan perilaku adalah “kemampuan” atau fitur dari objek tersebut.

Kita juga bisa definisikan dalam bentuk yang lebih sederhana lagi, yaitu: objek adalah sebuah gabungan dari kumpulan variabel (dinamakan atribut) dan kumpulan fungsi (dinamakan perilaku)[1]

Dan atas definisi itu, maka bisa dikatakan bahwa semua hal di dalam python adalah sebuah objek [2].

Bahkan Sebuah Fungsi Pun Adalah Objek

Kalau kita teliti lebih jauh lagi, setiap fungsi pada python memiliki atribut `__doc__` yang merupakan bukti bahwa fungsi sekalipun adalah sebuah objek dalam python.

Jika kita periksa tipe data dari sebuah fungsi, kita akan dapati bahwa ia ternyata objek dari sebuah `class` dengan nama `function`:

```
>>> def tes():
...     print('halo')
...
>>> type(tes)
<class 'function'>
```

Kelas Pada Python

Apa itu sebuah kelas?

Kelas atau *class* pada python bisa kita katakan sebagai sebuah *blueprint* (cetakan) dari objek (atau *instance*) yang ingin kita buat.

Kelas adalah cetakannya atau definisinya, sedangkan objek (atau *instance*) adalah objek nyatanya.

Kita coba beri contoh “kucing” untuk memperdekat pemahaman.

- “Kucing” merupakan sebuah definisi objek, ia memiliki 2 telinga, 4 kaki, 1 ekor, nama dan lain-lain (sebagai atribut), ia juga bisa berlari, mengeong, makan dan minum (sebagai perilaku).
- Misal kita memiliki 4 kucing: berarti kita memiliki “4 instance” dari kelas kucing.

- Masing-masing dari 4 kucing tersebut memiliki atribut yang telah didefinisikan sebelumnya seperti kaki, telinga, ekor, dan lain-lain. Ia juga memiliki kemampuan berlari, mengeong, dan sebagainya seperti yang telah didefinisikan sebelumnya.

Atribut dan Perilaku

Atribut pada OOP merepresentasikan variabel yang dimiliki sebuah objek.

Sedangkan perilaku pada OOP merepresentasikan fungsi yang dimiliki sebuah objek.

Contoh Atribut

Sebagai contoh, kita memiliki 3 ekor kucing, masing-masing kucing memiliki warna dan usianya sendiri-sendiri.

Jika kita ingin merepresentasikan 3 ekor kucing tersebut dengan pendekatan OOP, kita bisa menuliskan kode programnya seperti berikut:

```
# kelas kucing sebagai "definisi"
class Kucing:
  warna = None
  usia = None

# membangun instance/variabel sebagai "objek nyata"
kucing1 = Kucing()
kucing1.warna = "hitam"
kucing1.usia = "3 bulan"

kucing2 = Kucing()
kucing2.warna = "putih"
kucing2.usia = "2 bulan"

kucing3 = Kucing()
kucing3.warna = "kuning"
kucing3.usia = "3.5 bulan"
```

Kita bisa menampilkan masing-masing atribut dari tiap instance:

```
print(kucing1.warna, kucing1.usia)
print(kucing2.warna, kucing2.usia)
print(kucing3.warna, kucing3.usia)
```

Output:

```
hitam 3 bulan
putih 2 bulan
kuning 3.5 bulan
```

Contoh Perilaku

Misalkan kita memiliki kelas Mahasiswa. Setiap mahasiswa memiliki atribut:

1. Nama
2. dan Asal

Serta memiliki perilaku:

1. memperkenalkan diri

Maka bisa kita definisikan kode programnya dengan pendekatan OOP sebagai berikut:

```
# buat kelasnya terlebih dahulu
class Mahasiswa:
  nama = None
  asal = None

  def perkenalan (self):
    print(f'Perkenalkan saya {self.nama} dari {self.asal}')
```

Setelah membuat kelas, kita bisa membuat instance sesuka kita.

Dalam contoh berikut ini, saya akan membuat 2 buah mahasiswa: yang pertama mahasiswa dengan nama `"Deni"` dari Jawa Timur, dan yang kedua adalah `"Lendis Fabri"` yang juga dari Jawa Timur, kemudian saya akan minta keduanya untuk memperkenalkan diri.

Berikut ini kode programnya:

```
deni = Mahasiswa()
deni.nama = "Deni"
deni.asal = "Jawa Timur"

lendis = Mahasiswa()
lendis.nama = "Lendis Fabri"
lendis.asal = "Jawa Timur"

# panggil fungsi perkenalan
deni.perkenalan()
lendis.perkenalan()
```

Output:

```
Perkenalkan saya Deni dari Jawa Timur
Perkenalkan saya Lendis Fabri dari Jawa Timur
```

Kalau kita perhatikan lagi kode berikut:

```
def perkenalan (self):
  print(f'Perkenalkan saya {self.nama} dari {self.asal}')
```

Kita dapati ada sebuah parameter pada fungsi perkenalan, yaitu parameter `self`.

Tetapi ketika kita memanggil fungsi tersebut dari instance yang kita buat, kita tidak melemparkan parameter apa pun:

```
# panggil fungsi perkenalan
deni.perkenalan()
lendis.perkenalan()
```

Ini lah bedanya fungsi yang ada di luar kelas dan fungsi yang ada di dalam kelas.

Jadi ketika kita memanggil sebuah fungsi yang berada pada kelas tertentu, interpreter python akan otomatis melemparkan `self` sebagai parameter pertama. Dan dari paramter `self` ini kita bisa mendapatkan akses secara internal terhadap atribut dan perilaku dari objek yang kita buat.

Sebuah Instance dari Suatu Kelas

Kalau kita perhatikan lagi potongan kode program berikut:

```
deni = Mahasiswa()
lendis = Mahasiswa()
```

Pada 2 baris kode di atas, kita telah membuat 2 buah instance dari kelas `Mahasiswa`.

Dari segi penulisan, keduanya sama saja seperti cara penulisan variabel.

Karena sebenarnya *instance* itu ya variabel juga. Cuman bedanya, pada contoh di sini kita membuat “tipe data kita sendiri” dari pada menggunakan tipe data asli bawaan python.

Konstruktor

Apa itu konstruktor?

Konstruktor adalah sebuah fungsi yang akan dipanggil pertama kali saat sebuah objek di-*instantiasi*-kan.

Fungsi tersebut harus selalu bernama `__init__()`.

Agar lebih paham, mari kita praktikkan langsung untuk mendefinisikan fungsi konstruktor pada kelas `Mahasiswa`.

```
class Mahasiswa:

 def __init__(self, nama, asal):
```

```
    self.nama = nama

    self.asal = asal


  def perkenalan (self):
    print(f'Perkenalkan saya {self.nama} dari {self.asal}')
```

Ketika membuat instance dari kelas Mahasiswa, kita harus melemparkan dua buah parameter wajib yaitu parameter `nama` dan parameter `asal`, seperti ini:

```
deni = Mahasiswa('Deni', 'Sulawesi')
lendis = Mahasiswa(asal = 'Sumatera', nama = 'Lendis Fabri')

deni.perkenalan()
lendis.perkenalan()
```

Dokumentasi Kelas

Selain atribut dan perilaku, kelas juga memiliki “deskripsi”. Deskripsi ini didefinisikan dengan menuliskan komentar multi baris yang diletakkan langsung setelah pendefinisian fungsi.

Contohnya seperti ini:

```
class Mahasiswa:
  """
  Kelas ini digunakan untuk mendefinisikan
  objek Mahasiswa di kehidupan nyata
  """
```

Kita bisa menampilkannya dengan mengakses atribut bawaan yang bernama `__doc__` seperti berikut:

```
print(Mahasiswa.__doc__)
```

Output:

```
 Kelas ini digunakan untuk mendefinisikan
  objek Mahasiswa di kehidupan nyata
```

Atribut Kelas yang Merupakan Instance dari Kelas Lainnya

Terakhir dan bukan yang paling akhir, kita juga bisa mendefinisikan sebuah atribut objek yang merupakan *instance* dari objek yang lainnya.

Misal: kita jadikan kelas Mahasiswa memiliki satu ekor kucing.

Kode programnya bisa seperti ini:

```
class Kucing:
  def __init__(self, warna, usia):
    self.warna = warna
    self.usia = usia

class Mahasiswa:
  def __init__(self, nama, asal, kucing):
    self.nama = nama
    self.asal = asal
    self.kucing = kucing

  def perkenalan(self):
    print(f'Perkenalkan saya {self.nama} dari {self.asal}')
    print(f'Saya memiliki kucing berwarna {self.kucing.warna} usia
{self.kucing.usia}')
```

Cara memanggilnya:

```
deni = Mahasiswa(
    nama='Deni',
    asal='Sidoarjo',
    kucing=Kucing(
```

```
        warna='Merah',
        usia='3 bulan'
    )
)

deni.perkenalan()
```

Output:

```
Perkenalkan saya Deni dari Sidoarjo
Saya memiliki kucing berwarna Merah usia 3 bulan
```

Kesimpulan

1. Objek pada python adalah kumpulan dari variabel-variabel (dinamakan atribut) dan kumpulan dari fungsi-fungsi (dinamakan perilaku).
2. Atas definisi itu, maka semua hal di dalam python adalah sebuah Objek.
3. Objek dan Kelas dalam python bermakna sama. Akan tetapi, jika disebutkan dalam konteks terpisah, maka kelas adalah *blueprint* dan objek adalah varibel nyata.
4. Konstruktor adalah fungsi yang pertama kali dipanggil ketika sebuah objek diinstantiasi.
5. Objek bisa memiliki atribut yang berupa instan dari kelas lainnya.

Python: Pewarisan (Inheritance) 🐍

Memahami Konsep Pewarisan Objek

Bagaimana konsep pewarisan bekerja?

Konsep pewarisan adalah konsep di mana sebuah kelas atau objek mewariskan sifat dan perilaku kepada kelas lainnya.

Kelas yang menjadi “pemberi waris” dinamakan kelas induk atau kelas basis

Sedangkan kelas yang menjadi “ahli waris” dinamakan sebagai kelas turunan.

Sifat Pewarisan

Secara umum, kelas turunan akan selalu memiliki sifat dan perilaku yang sama dengan kelas induknya: mulai dari atribut sampai fungsi-fungsinya.

Akan tetapi tidak sebaliknya, belum tentu kelas induk memiliki semua atribut dan sifat dari kelas-kelas turunannya.

Contoh Sederhana Penerapan Pewarisan Objek

Untuk mempermudah pemahaman, mari kita buat contoh kasus sederhana yang akan kita selesaikan dengan konsep pewarisan.

Kita akan membuat 3 buah objek:

- Orang
- Pelajar
- Pekerja

Masing-masing dari 3 objek tersebut memiliki beberapa sifat dan perilaku yang sama, misalkan:

- `nama`
- `asal`
- dan kemampuan `memperkenalkan diri`

Tetapi, sebagian objek tetap memliki ciri khasnya masing-masing, misalkan:

- Objek `Pelajar` memiliki atribut `sekolah`.
- Objek `Pekerja` memiliki atribut `tempat kerja`.

Cari hubungan “pewarisan” antar ketiganya

Untuk menyelesaikan kasus di atas, kita perlu menarik sebuah premis bahwa ketiganya memiliki hubungan “waris”.

Seperti apa hubungan tersebut?

Hubungannya adalah:

- Objek `Pelajar` sebenarnya adalah objek `Orang` juga, hanya saja objek `Pelajar` memiliki atribut tambahan yang tidak dimiliki objek `Orang`.
- Begitu pula objek `Pekerja`, ia sebenarnya adalah objek `Orang` juga, hanya saja ia memiliki atribut tambahan yang tidak dimiliki objek `Orang`.

Definisikan Kelas Basis dan Kelas Turunan

Setelah mengetahui gambaran hubungan “waris” antar ketiga keelas tersebut, kita bisa simpulkan bahwa:

- Objek `Pelajar` dan objek `Pekerja` adalah kelas turunan dari objek `Orang`.

Setelah berhasil menentukan mana objek induk dan mana objek turunan, mari kita tuangkan hal tersebut kedalam kode program 😎

Membuat Objek Induk (Parent)

Untuk membuat objek induk pada python, caranya sama dengan cara membuat objek biasa. Karena pada dasarnya, semua objek pada python bisa menjadi objek induk dari objek turunan lainnya.

Mari kita buat objek `Orang` seperti skenario yang telah kita bahas pada bagian sebelumnya:

```
class Orang:

  def __init__ (self, nama, asal):
    self.nama = nama
    self.asal = asal
```

```
  def perkenalan (self):
    print(f'Perkenalkan nama saya {self.nama} dari {self.asal}')
```

Objek di atas sangat sederhana, ia hanya memiliki 2 atribut (`nama` dan `asal`) serta memiliki satu buah perilaku yaitu `perkenalan()`.

Kita bisa membuat instance dari kelas `Orang` seperti biasanya:

```
andi = Orang('Andi', 'Surabaya')
andi.perkenalan()
```

Output:

```
Perkenalkan nama saya Andi dari Surabaya
```

Membuat Objek Turunan (Child)

Langkah selanjutnya adalah: membuat objek turunan dari kelas `Orang`. Sesuai dengan skenario yang telah kita bahas di atas, kita akan membuat dua buah objek baru yaitu objek `Pelajar` dan `Pekerja`, yang mana keduanya akan mewarisi objek `Orang`.

Caranya bagaimana?

Kita bisa membuat kelas turunan dengan cara mengirimkan kelas induk sebagai parameter saat mendefinisikan kelas.

Perhatikan contoh berikut:

```
class Pelajar (Orang):
  pass

class Pekerja (Orang):
  pass
```

Kita telah membuat dua buah kelas yang keduanya sama-sama memiliki setiap atribut dan fungsi dari kelas `Orang`.

Kita meletakkan perintah pass karena kita hanya ingin melakukan pewarisan apa adanya tanpa menambahkan apa pun lagi.

Sehingga, kita bisa membuat instance dari kelas `Pelajar` dan `Pekerjaan`, serta memanggil fungsi `perkenalan()` dengan cara yang benar-benar identik dengan kelas `Orang`:

```
andi = Orang('Andi', 'Surabaya')

andi.perkenalan()


deni = Pelajar('Deni', 'Makassar')

deni.perkenalan()


budi = Pekerja('Budi', 'Pontianak')

budi.perkenalan()
```

Output:

```
Perkenalkan nama saya Andi dari Surabaya
Perkenalkan nama saya Deni dari Makassar
Perkenalkan nama saya Budi dari Pontianak
```

Membuat Konstruktor Pada Kelas Turunan

Konstruktor pada kelas turunan memiliki perilaku yang sedikit berbeda dengan konstruktor yang terdapat pada kelas induk.

Apa yang terjadi ketika kelas turunan memiliki konstruktor sendiri?

Ia akan menimpa konstruktor dari kelas induk sehingga konstruktor kelas induk tidak akan pernah dieksekusi.

Coba kita ubah konstruktor pada kelas `Orang` menjadi seperti berikut:

```
class Orang:

  def __init__ (self, nama, asal):
    self.nama = nama
    self.asal = asal

   print('fungsi Orang.__init__() dieksekusi')
```

Jalankan kembali program kita, kita akan mendapatkan output seperti berikut:

```
fungsi Orang.__init__() dieksekusi
Perkenalkan nama saya Andi dari Surabaya
fungsi Orang.__init__() dieksekusi
Perkenalkan nama saya Deni dari Makassar
fungsi Orang.__init__() dieksekusi
Perkenalkan nama saya Budi dari Pontianak
```

Sekarang, kita tambahkan konstruktor pada masing-masing kelas `Pelajar` dan `Pekerja`.

```
def __init__ (self, nama, asal):
  self.nama = nama
  self.asal = asal
```

Jalankan kembali kode program kita, dan kita akan mendapatkan output yang berbeda:

```
fungsi Orang.__init__() dieksekusi
```

```
Perkenalkan nama saya Andi dari Surabaya
Perkenalkan nama saya Deni dari Makassar
Perkenalkan nama saya Budi dari Pontianak
```

Sangat jelas dari output di atas, bahwa konstruktor kelas `Orang` tidak lagi dieksekusi ketika kita membuat instance dari kelas `Pelajar` dan `Pekerja`. Hal itu terjadi karena sekarang kedua kelas tersebut telah memiliki konstruktor sendiri.

Fungsi `super().__init__()` atau `KelasInduk.__init__()`

Menimpa konstruktor kelas induk adalah ide yang buruk. Karena hal tersebut akan menghilangkan sebagian dari “pewarisan” itu sendiri. Di sisi lain, menambahkan fungsi konstruktor ke dalam kelas turunan ternyata justru menimpa fungsi konstruktor milik kelas induk.

Lalu bagiamana solusisnya?

Solusinya adalah dengan memanggil fungsi konstruktor pada kelas induk secara implisit.

Caranya ada 2:

- yang pertama menggunakan fungsi `super().__init__()`
- dan yang kedua menggunakan `KelasInduk.__init__()`

Apa beda keduanya?

Untuk cara pertama: `super().__init__()` kita cukup memanggil fungsinya seperti biasanya, tidak perlu mendefinisikan lagi data `self`.

Ada pun untuk cara yang kedua, kita harus mengirimkan data `self` secara manual.

Agar lebih jelas, mari kita langsung praktikkan keduanya.

Kita coba cara yang pertama untuk kelas `Pelajar`, dan cara yang kedua untuk kelas `Pekerja`:

```
class Pelajar (Orang):
  def __init__ (self, nama, asal):

    super().__init__(nama, asal)


class Pekerja (Orang):
  def __init__ (self, nama, asal):

    Orang.__init__(self, nama, asal)
```

Jalankan kembali aplikasi, dan kita akan mendapatkan output sebagai berikut:

```
fungsi Orang.__init__() dieksekusi
Perkenalkan nama saya Andi dari Surabaya
fungsi Orang.__init__() dieksekusi
Perkenalkan nama saya Deni dari Makassar
fungsi Orang.__init__() dieksekusi
Perkenalkan nama saya Budi dari Pontiana
```

Sekarang fungsi konstruktor dari kelas induk (yaitu kelas `Orang`) tetap terpanggil dari kelas turunannya.

Menambahkan Properti Baru Yang Tidak Ada Pada Induk

Seperti yang telah berlalu penjelasannya:

Bahwa kelas turunan akan memiliki semua sifat dan perilaku dari kelas induk, akan tetapi belum tentu kelas induk memiliki semua sifat dan perilaku dari kelas turunannya.

Itu artinya, kita bisa menambahkan atribut tertentu atau fungsi tertentu pada kelas turunan yang tidak ada pada kelas induk.

Bagaimana caranya?

Ya tinggal tambahkan saja, sederhana sekali 😁

Perhatikan contoh berikut, kita akan menambahkan atribut `sekolah` untuk kelas `Pelajar`, dan atribut `tempat_kerja` untuk kelas `Pekerja`:

```
class Pelajar (Orang):
  def __init__ (self, nama, asal, sekolah):
    super().__init__(nama, asal)

    self.sekolah = sekolah


class Pekerja (Orang):
  def __init__ (self, nama, asal, tempat_kerja):
    Orang.__init__(self, nama, asal)

    self.tempat_kerja = tempat_kerja
```

Dan terakhir, kita sesuaikan kode ketika membuat instance dari 2 kelas tersebut:

```
deni = Pelajar('Deni', 'Makassar', 'SMA Negeri 1 Makassar')

deni.perkenalan()

print(f'Saya sekolah di {deni.sekolah}')



budi = Pekerja('Budi', 'Pontianak', 'Google')

budi.perkenalan()

print(f'Saya bekerja di {budi.tempat_kerja}')
```

Oiya, sebelumnya kalian juga bisa menghapus kode yang menampilkan output `fungsi Orang.__init__() dieksekusi` karena kita sudah tidak membutuhkannya lagi.

Sehingga output akhir dari pertemuan kali ini adalah:

```
Perkenalkan nama saya Andi dari Surabaya
Perkenalkan nama saya Deni dari Makassar
Saya sekolah di SMA Negeri 1 Makassar
Perkenalkan nama saya Budi dari Pontianak
Saya bekerja di Google
```

Python: Overriding (Penimpaan)

Pengertian Overriding

Apa itu overriding dalam konsep pemrograman berorientasi objek?

Di dalam semua bahasa pemrograman yang berbasis objek, teknik *overriding* adalah fitur yang memungkinkan kita untuk mengimplementasikan “ulang” fungsi/method pada sebuah *child class* atau kelas turunan yang sebenarnya fungsi tersebut telah didefinisikan di dalam *parent class* atau kelas induk [1].

Overriding sendiri memberikan keuntungan kepada kita karena kita bisa menduplikat kelas lain –*sehingga kode program lebih singkat*, dan kita juga bisa “membedakan” fungsi-fungsi tertentu pada kelas turunan yang sudah didefinisikan pada kelas Induk.

Hal ini memberikan kita keleluasaan untuk mendefinisikan berbagai kelas sesuai fungsinya masing-masing tanpa harus menulis kode yang sama berkali-kali. Oleh karena itu, *overriding* masih dikategorian sebuah bagian dari *pewarisan* [2].

Cara Melakukan Overriding Pada Python

Untuk melakukan *overriding* pada python caranya sederhana:

1. Kita hanya perlu melakukan pewarisan.
2. Kemudian menulis ulang fungsi yang sudah ada pada kelas Induk.

Contoh Overriding

Misalkan kita punya dua buah kelas yaitu `Kendaraan` dan `Mobil` di mana:

- Kelas `Kendaraan` adalah kelas induk
- dan Kelas `Mobil` adalah kelas turunan dari kelas `Kendaraan`

Dan:

- Kelas `Kendaraan` punya kemampuan (fungsi) `berjalan()`
- Akan tetapi kita ingin bahwa kelas `Mobil` punya perilaku khusus untuk fungsi `berjalan()`.

Agar lebih jelas, mari kita praktikkan secara langsung.

Sebagai pondasi awal, buat dua buah kelas seperti berikut:

```
class Kendaraan:
  def berjalan(self):
    print('berjalan..')

class Mobil(Kendaraan):
  pass
```

Kemudian buat instan dan panggil fungsi `berjalan()`:

```
sepeda = Kendaraan()
sedan = Mobil()

sepeda.berjalan()
sedan.berjalan()
```

Output:

```
berjalan..
berjalan..
```

Jika kita perhatikan lagi output di atas: maka output-nya identik, tidak ada yang berbeda satu huruf pun. Itu karena kelas `Mobil` menurunkan semua hal dari kelas `Kendaraan` tanpa mengubah satu hal apa pun.

Melakukan Overriding

Nah, langkah berikutnya adalah kita akan melakukan *overriding* fungsi `berjalan()`, agar aksinya menjadi berbeda ketika dipanggil dari kelas turunan yaitu kelas `Mobil`.

Ubah kode programnya menjadi seperti ini:

```
class Kendaraan:
  def berjalan(self):
    print('berjalan..')

class Mobil(Kendaraan):

 def berjalan(self):

   print('Berjalan dengan cepat..')
```

Jalankan kembali programnya, maka kita akan mendapatkan output seperti ini:

```
berjalan..
Berjalan dengan cepat..
```

Menambahkan Parameter Pada Fungsi yang Ditimpa

Kita juga bisa memberikan parameter pada fungsi yang ingin kita timpa. Sebagai contoh, kita akan menambahkan 2 buah parameter untuk fungsi `berjalan()` pada kelas `Mobil`.

Dua buah parameter tersebut adalah:

- kecepatan
- satuan kecepatan

Berikut ini kode programnya:

```
class Kendaraan:
  def berjalan(self):
    print('berjalan..')

class Mobil(Kendaraan):

 def berjalan(self, kecepatan, satuan = 'km/j'):

   print(f'Berjalan dengan kecepatan {kecepatan} {satuan}')
```

Jangan lupa ubah juga pemanggilan fungsi `berjalan()` dengan menambahkan parameter yang sesuai:

```
sepeda = Kendaraan()
sedan = Mobil()

sepeda.berjalan()

sedan.berjalan(150)
```

Output:

```
berjalan..
Berjalan dengan kecepatan 150 km/j
```

Memanggil Fungsi Pada Kelas Induk

Sama seperti yang telah kita pelajari pada pertemuan sebelumnya tentang fungsi super dalam konstruktor (masih dalam bab pewarisan), kita memanggil fungsi konstruktor yang terdapat pada kelas induk dari kelas turunan dengan sintaks berikut:

```
class KelasTurunan(KelasInduk):
  def __init__(self):

    super().__init__()
```

Di mana sebenarnya fungsi `super()` tidak khusus hanya untuk konstruktor saja, tapi kita juga bisa menggunakannya untuk memanggil fungsi selain konstruktor.

Perhatikan contoh berikut:

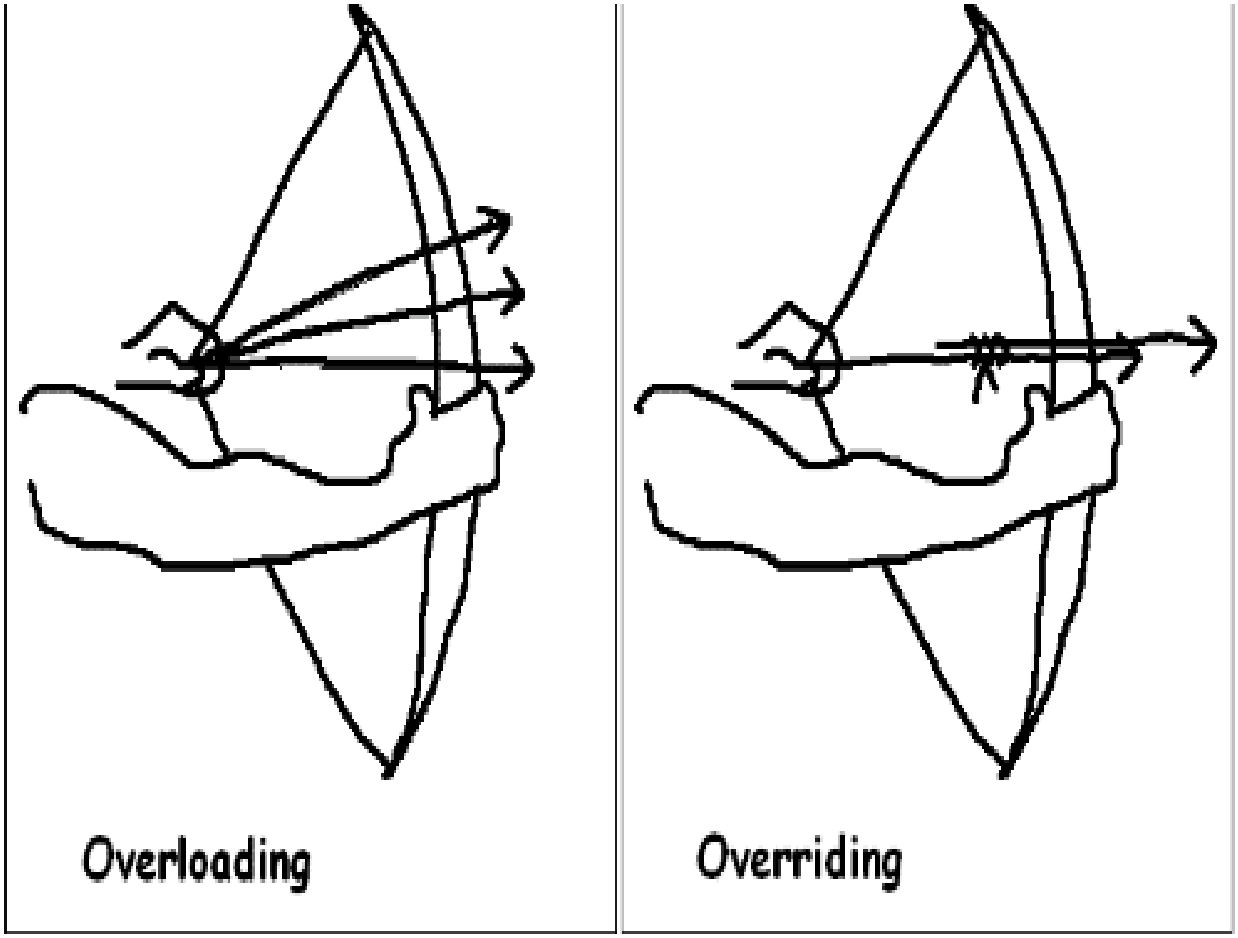
```
class Kendaraan:
  def berjalan(self):
    print('berjalan..')

class Mobil(Kendaraan):
  def berjalan(self, kecepatan, satuan = 'km/j'):

    super().berjalan()

    print(f'  -> dengan kecepatan {kecepatan} {satuan}')
```

Pada kode di atas, kita telah mendefinisikan ulang fungsi `berjalan()` pada kelas `Mobil`. Tidak hanya itu, kita juga tetap memanggil fungsi yang sama yang terdapat pada kelas induknya yaitu kelas `Kendaraan`.



Sehingga jika kita jalankan program di atas, kita akan mendapatkan output seperti ini:

```
berjalan..
berjalan..
  -> dengan kecepatan 150 km/j
```

Kesimpulan

- Teknik *overriding* merupakan bagian dari teknik pewarisan.
- Teknik *overriding* membuat kita bisa memodifikasi fungsi yang sudah didefinisikan pada kelas Induk
- Kita juga bisa mempertahankan fungsi asli yang terdapat pada kelas induk dengan memanggil fungsi `super()`

Python: Access Modifiers (Atau Enkapsulasi)

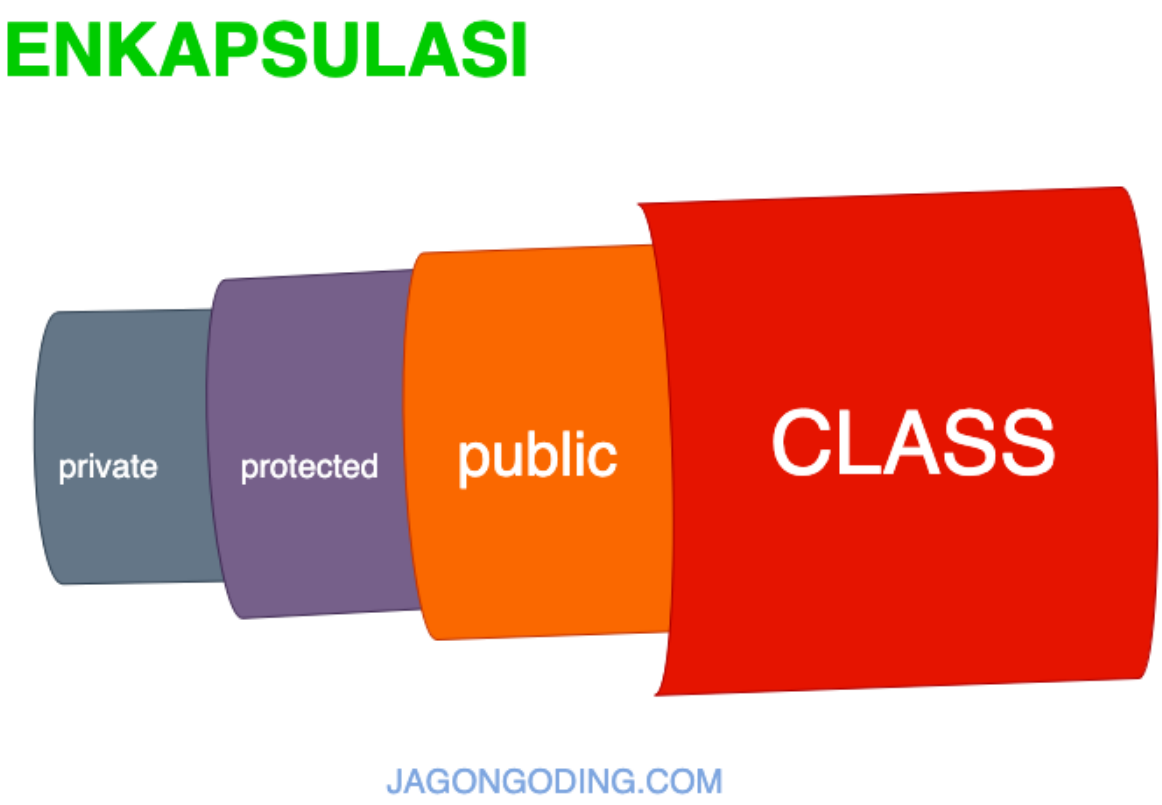
Access Modifiers adalah sebuah konsep di dalam pemrograman berorientasi objek di mana kita bisa mengatur hak akses suatu atribut dan fungsi pada sebuah class. Konsep ini juga biasa disebut sebagai enkapsulasi, di mana kita akan mendefinisikan mana atribut atau fungsi yang boleh diakses secara terbuka, mana yang bisa diakses secara terbatas, atau mana yang hanya bisa diakses oleh internal kelas alias privat.

Jenis Access Modifiers Pada Python

Sama seperti bahasa pemrograman berbasis objek lainnya semisal java, c++, php, dan lain-lain, python juga mendukung fitur access modifiers.

Access modifiers yang tersedia di dalam python ada 3:

1. Public
2. Protected
3. dan Private



Public Access Modifier

Variabel atau atribut yang memiliki hak akses publik bisa diakses dari mana saja baik dari luar kelas mau pun dari dalam kelas.

Perhatikan contoh berikut:

```
class Segitiga:

  def __init__(self, alas, tinggi):
    self.alas = alas
    self.tinggi = tinggi
```

```
    self.luas = 0.5 * alas * tinggi
```

Pada contoh di atas, kita membuat sebuah kelas dengan nama `Segitiga`. Kelas tersebut menerima dua buah parameter: yaitu `alas` dan `tinggi`.

Kemudian ia juga memiliki 3 buah atribut (`alas`, `tinggi`, dan `luas`) yang mana semuanya memiliki hak akses publik.

Untuk membuktikannya, mari kita coba akses ke-3 atribut di atas dari luar kelas:

```
segitiga_besar = Segitiga(100, 80)

# akses variabel alas, tinggi, dan luas dari luar kelas
print(f'alas: {segitiga_besar.alas}')
print(f'tinggi: {segitiga_besar.tinggi}')
print(f'luas: {segitiga_besar.luas}')
```

Jika dijalankan, kita akan mendapatkan output seperti berikut:

```
alas: 100
tinggi: 80
luas: 4000.0
```

Protected Access Modifier

Variabel atau atribut yang memiliki hak akses protected hanya bisa diakses secara terbatas oleh dirinya sendiri (yaitu di dalam internal kelas), dan juga dari kelas turunannya.

Mari kita mencoba mempraktikkannya.

Untuk mendefinisikan atribut dengan hak akses *protected*, kita harus menggunakan prefix underscore `_` sebelum nama variabel.

```
class Mobil:
```

```
  def __init__(self, merk):

    self._merk = merk
```

Pada kode program di atas, kita membuat sebuah kelas bernama `Mobil`. Dan di dalam kelas tersebut, kita mendefinisikan satu buah atribut bernama `_merk`, yang mana hak akses dari atribut tersebut adalah *protected* karena nama variabelnya diawali oleh *underscore* (`_`).

Mari kita coba akses dari luar kelas:

```
sedan = Mobil('Toyota')

# tampilkan _merk dari luar kelas
print(f'Merk: {sedan._merk}')
```

Ya, betul sekali. Kita masih bisa mengakses atribut `_merk` dari luar kelas, karena hal ini hanya bersifat *convention* alias adat atau kebiasaan saja yang harus disepakati oleh programmer [1]. Di mana jika suatu atribut diawali oleh `_`, maka ia harusnya tidak boleh diakses kecuali dari internal kelas tersebut atau dari kelas yang mewarisinya.

Yang harusnya kita lakukan adalah: mengakses atribut *protected* hanya dari kelas internal atau dari kelas turunan, perhatikan contoh berikut:

```
class Mobil:
  def __init__(self, merk):
    self._merk = merk

class MobilBalap(Mobil):
  def __init__(self, merk, total_gear):
    super().__init__(merk)
    self._total_gear = total_gear

  def pamer(self):
    # akses _merk dari subclass
    print(
      f'Ini mobil {self._merk} dengan total gear {self._total_gear}'
    )
```

Bikin objek dari kelas `MobilBalap`:

```
ferrari = MobilBalap('Ferrari', 8)
ferrari.pamer()
```

Output:

```
Ini mobil Ferrari dengan total gear 8
```

Private Access Modifier

Modifier selanjutnya adalah *private*. Setiap variabel di dalam suatu kelas yang memiliki hak akses *private* maka ia hanya bisa diakses di dalam kelas tersebut. Tidak bisa diakses dari luar bahkan dari kelas yang mewarisinya.

Untuk membuat sebuah atribut menjadi *private*, kita harus menambahkan dua buah underscore sebagai *prefix* nama atribut.

Perhatikan contoh berikut:

```
class Mobil:
  def __init__(self, merk):

    self.__merk = merk
```

Pad kode di atas, kita telah membuat sebuah atribut dengan nama `__merk`. Dan karena nama tersebut diawali dua buah underscore, maka ia tidak bisa diakses kecuali dari dalam kelas `Mobil` saja.

Mari kita buktikan:

```
jip = Mobil('Jeep')
print(f'Merk: {jip.__merk}')
```

Kita akan mendapatkan sebuah error sebagai berikut:

```
AttributeError: 'Mobil' object has no attribute '__merk'
```

Tapi jika kita ubah kodenya menjadi seperti ini:

```
class Mobil:
  def __init__(self, merk):
    self.__merk = merk

  def tampilkan_merk(self):

    print(f'Merk: {self.__merk}')


jip = Mobil('Jeep')

jip.tampilkan_merk()
```

Kita akan mendapatkan output:

```
Merk: Jeep
```

Kenapa? Karena kita mengakses variabel `__merk` dari dalam internal kelas `Mobil`, bukan dari luar.

Accessor Mutator

Selanjutnya kita bisa membuat accessor dan mutator atau getter dan setter pada suatu kelas di python.

Untuk apa accessor dan mutator? Ia adalah sebuah fungsi yang akan dieksekusi ketika kita mengakses (aksesor) suatu atribut pada suatu kelas, atau fungsi yang dieksekusi ketika hendak mengatur (mutator) suatu atribut pada suatu kelas.

Untuk mendefinisikan accessor (getter), kita perlu mendefinisikan decorator `@property` sebelum nama fungsi.

Sedangkan untuk mengatur mutator (setter), kita perlu mendefinisikan descriptor `@<nama-atribut>.setter`.

Perhatikan contoh berikut:

```
class Mobil:
  def __init__(self, tahun):
    self.tahun = tahun


 @property

 def tahun(self):
    return self.__tahun


 @tahun.setter

 def tahun(self, tahun):
    if tahun > 2021:
      self.__tahun = 2021
    elif tahun < 1990:
      self.__tahun = 1990
    else:
      self.__tahun = tahun
```

Ketika kita nanti mengakses atribut `tahun` (tanpa underscore), maka fungsi `tahun()` yang pertama akan dieksekusi.

Sedangkan jika kita mengisi / mengubah / memberi nilai pada atribut `tahun` (tanpa underscore), maka yang dieksekusi adalah fungsi `tahun()` yang kedua.

Mari kita coba kode program berikut:

```
sedan = Mobil(2200)
print(f'Mobil ini dibuat tahun {sedan.tahun}')
```

Kode program di atas akan menampilkan tahun dari sedan yang sudah melalui `if-else` sebelumnya. Sehingga jika kita memberikan nilai `tahun` yang lebih dari tahun `2021`, yang tersimpan tetaplah tahun `2021`.

Berikut ini outputnya:

```
Mobil ini dibuat tahun 2021
```

Sedangkan jika kita mengakses atribut aslinya yaitu `__tahun`, kita akan mendapatkan error karena atribut tersebut bersifat *private*.

```
print(f'Mobil ini dibuat tahun {sedan.__tahun}')
```

Error:

```
AttributeError: 'Mobil' object has no attribute '__tahun'
```

Selanjutnya mari kita coba ubah atribut `tahun` seperti berikut:

```
sedan = Mobil(2000)
sedan.tahun = 1800
print(f'Mobil ini keluaran {sedan.tahun}')
```

Output:

```
Mobil ini keluaran 1990
```

Lihat, kita mengubah tahun menjadi `1800` akan tetapi yang tersimpan adalah `1990`. Itu terjadi karena ketika kita mengubah atribut `tahun`, sistem masih memanggil terlebih dahulu fungsi setter `tahun()`.

Menggabungkan Public, Protected, dan Private

Selain itu, kita juga bisa menggunakan 3 buah modifiers sekaligus dalam satu kelas, perhatikan contoh berikut:

```
class SebuahKelas:
  def __init__(self):
    self.publik = 'Atribut Publik'
    self._protected = 'Atribut Protected'
    self.__privat = 'Atribut Privat'
```

Pada contoh di atas kita memiliki 3 buah atribut:

1. Atribut `publik` dengan modifier *public*
2. Atribut `_protected` dengan modifier *protected*
3. Dan atribut `__privat` dengan modifier *private*

Kesimpulan

Dalam pemrograman python, kita bisa memanfaatkan fitur access modifier untuk mengenkapsulasi sebuah kode program. Alias mengatur mana atribut yang boleh diakses dari luar, dan mana atribut yang hanya konsumsi internal.

Sayangnya, kelemahan *access modifier* dalam python adalah:

- Tidak ada atribut yang benar-benar *public* atau *protected*. Selama sebuah variabel tidak diawali dua buah *underscore*, maka ia bisa diakses dari mana pun.

- Hal tersebut menjadikan konsep “protected” modifier hanyalah sebuah *convention* atau kebiasaan saja, di mana kalau kita mendefinisikan sebuah atribut yang diawali satu buah underscore, maka kita seharusnya tidak mengakses atribut tersebut kecuali dari dalam kelas itu sendiri atau dari kelas turunannya.

Python: Operator Overloading

Pengertian Overloading

Overloading –dalam dunia pemrograman– adalah teknik untuk mengatur berbagai perilaku dari suatu fungsi berdasarkan dengan parameter yang diterimanya, atau perilaku objek berdasarkan dengan operator yang sedang dioperasikan [1]. Satu fungsi atau satu objek bisa memiliki perilaku yang berbeda-beda tergantung dengan kondisi yang ia terima.

Pada pertemuan ini, kita akan lebih fokus terhadap operator overloading.

Apa itu Operator Overloading?

Operator overloading adalah teknik di mana kita akan mengatur atau mendefinisikan perilaku sebuah kelas (yang kita buat), bagaimana ia akan berinteraksi dengan berbagai macam operator yang berbeda.

Kita ambil contoh operator `+`.

Operator `+` di dalam python memiliki lebih dari satu peran tergantung tipe data dari operan yang diterima.

Misal kita operasikan operator `+` terhadap nilai `int`, ia akan melakukan operasi penjumlahan.

Contoh:

```
>>> 10 + 20
30
>>>
```

Dan jika kita operasikan terhadap dua buah string atau lebih, ia akan melakukan operasi "penggabungan".

Perhatikan contoh berikut:

```
>>> 'Indo' + 'nesia'
'Indonesia'
>>>
```

Hal ini karena operator `+` telah di-"overloaded" di kelas `int` dan kelas `str` sehingga memiliki perilaku yang berbeda.

Membuat Operator Overloading

Nah, pertanyaannya adalah: bagaimana kita bisa menentukan aksi suatu kelas jika kelas yang kita buat tersebut dijadikan sebagai operan dari suatu operator?

Misalkan kita memiliki kelas seperti berikut:

```
class Angka:
  def __init__(self, angka):
    self.angka = angka
```

Lalu kita ingin menambahkan dua buah instance dari kelas `Angka`:

```
>>> x1 = Angka(10)
>>> x2 = Angka(20)
>>>
>>> x1 + x2
```

Yang ada justru kita mendapatkan error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Angka' and 'Angka'
```

Error di atas menyatakan kalau kelas `Angka` tidak bisa menjadi operand dari operator `+`.

Untuk membuatnya, kita harus mendefinisikan *magic method* bernama `__add__()` seperti berikut:

```
class Angka:
  def __init__(self, angka):
    self.angka = angka

  def __add__(self, objek):
    return self.angka + objek.angka
```

Ketika kita panggil:

```
x1 = Angka(10)
x2 = Angka(20)

print(x1 + x2)
```

Kita akan mendapatkan output:

```
30
```

Atau kita juga bisa me-return instan baru dari kelas `Angka` agar hasilnya tetap konsisten seperti berikut:

```
class Angka:
  def __init__(self, angka):
    self.angka = angka

  def __add__(self, objek):
    return Angka(
```

```
        self.angka + objek.angka
    )
```

Ketika dipanggil:

```
x1 = Angka(5)
x2 = Angka(20)
x3 = x1 + x2

print(x3.angka)
```

Output:

```
25
```

Daftar Fungsi Untuk Operator Aritmatika

Selain fungsi `__add__` untuk menangani operator `+`, kita juga bisa menggunakan berbagai fungsi lain untuk menangani berbagai operator.

Lebih jelasnya silakan perhatikan tabel berikut:

Operator	Nama Fungsi
+	`__add__()`
-	`__sub__()`
*	`__mul__()`
/	`__truediv__()`

//	`__floordiv__()`
%	`__mod __()`
**	`__pow__()`
»	`__rshift__()`
«	`__lshift__()`
&	`__and__()`
|	`__or__()`
^	`__xor__()`

Menangani Operator Perbandingan

Selain operator aritmatika, python juga menyediakan magic method untuk menangani operator perbandingan seperti operator lebih dari (`>`), kurang dari (`<`), operator sama dengan (`==`) dan lain sebagainya.

Untuk operator perbandingan lebih dari, kita bisa menggunakan fungsi `__gt__` yang merupakan singkatan dari *greater than*.

Untuk operator kurang dari, kita bisa menggunakan fungsi `__lt__` yang merupakan singkatan dari *less than*.

Dan untuk operator sama dengan, kita bisa menggunakan fungsi `__eq__` yang merupakan singkatan dari *equal*.

Perhatikan contoh berikut:

```
class Angka:
  def __init__(self, angka):
    self.angka = angka

  def __gt__(self, objek):
    return self.angka > objek.angka

  def __lt__(self, objek):
    return self.angka < objek.angka

  def __eq__(self, objek):
    return self.angka == objek.angka
```

Kemudian kita bisa membandigkan antar dua objek dari kelas `Angka` seperti berikut:

```
x1 = Angka(20)
x2 = Angka(10)

print(x1 > x2)
print(x1 < x2)
print(x1 == x2)
```

Output:

```
True
False
False
```

Daftar Fungsi Untuk Operator Komparasi

Untuk daftar lengkap nama fungsi + operator komparasi yang sesuai, silakan perhatikan tabel berikut:

Operator	Nama Fungsi

<	`__lt__()`
>	`__gt__()`
<=	`__le__()`
>=	`__ge__()`
==	`__eq__()`
!=	`__ne__()`

Daftar Fungsi Untuk Operator Assignment

Selain operator aritmatika seperti penjumlahan (`+`) dan pengurangan (`-`), kita juga bisa melakukan operator overloading untuk operator *shortcut assignment*.

Perhatikan tabel berikut:

Operator	Nama Fungsi
-=	`__isub__()`
+=	`__iadd__()`
*=	`__imul__()`
/=	`__idiv__()`
//=	`__ifloordiv__()`

%=	`__imod__()`
**=	`__ipow__()`
»=	`__irshift__()`
«=	`__ilshift__()`
&=	`__iand__()`
|=	`__ior__()`
^=	`__ixor__()`

Kesimpulan

Setelah mempelajari konsep OOP pada python, kita jadi tahu bahwa semua hal pada python adalah objek. Bahkan tipe data asli seperti `int` dan `str` pun juga termasuk objek.

Hal itu membuat kita –sebagai programmer– bisa membuat tipe data kita sendiri.

Dan setelah membuat tipe data sendiri, kita bisa melakukan teknik operator overloading untuk mengatur bagaimana objek yang kita buat akan bereaksi terhadap suatu operator tertentu.

Teknik operator overloading sendiri bisa kita implementasikan untuk berbagai operator: di antaranya operator aritmatika, perbandingan, dan juga operator penugasan (*assignment*).