**Duck Typing**

Python sering dikaitkan dengan metode *duck typing*, yakni metode ini berbunyi seperti berikut.



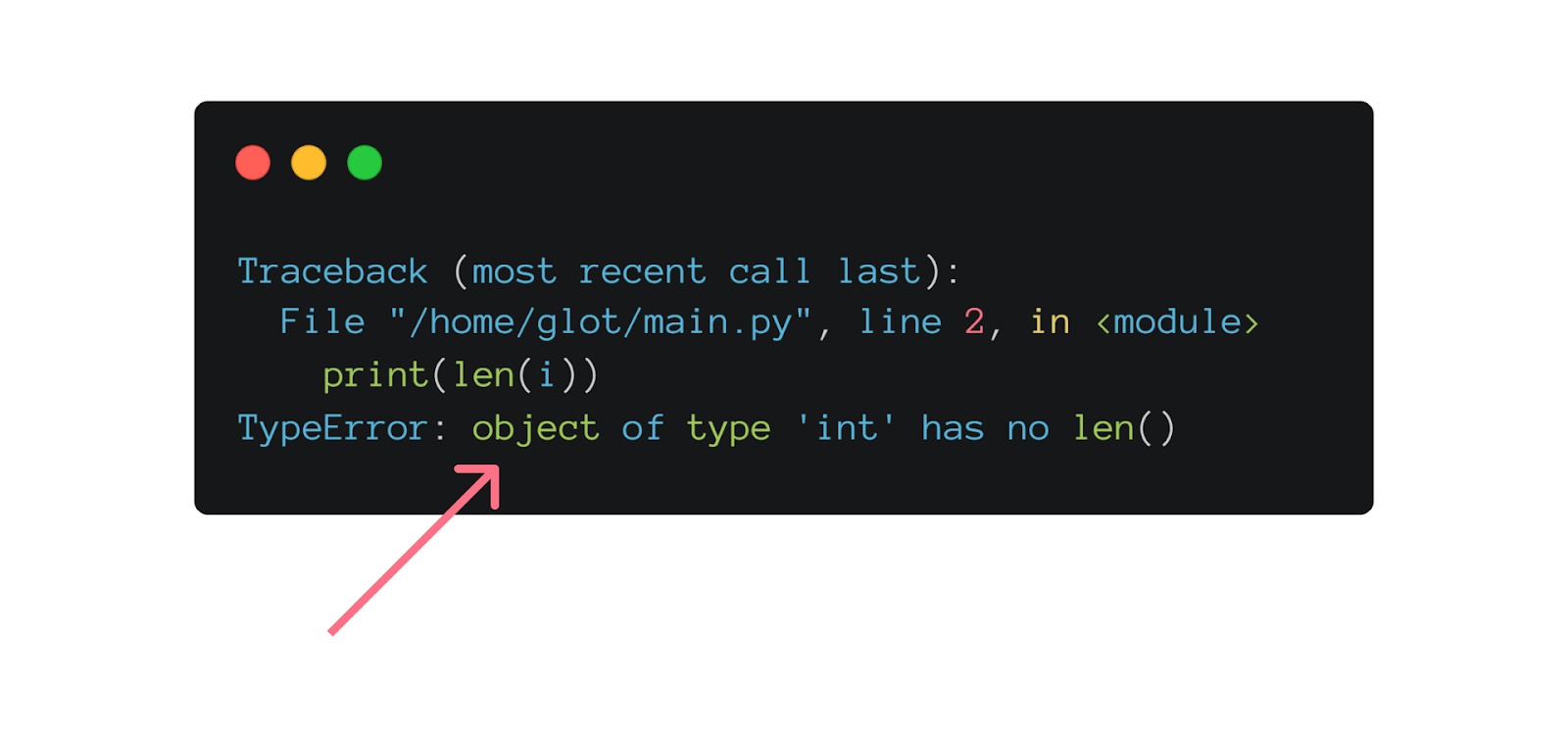
Konsep duck typing tidak berkaitan langsung dengan *dynamic typing* atau *loosely typed*, ini merupakan konsep yang lebih erat dengan materi kita kali ini, yaitu pemrograman berorientasi objek (*object-oriented programming*). Duck typing menjelaskan bahwa sebuah tipe atau *class* dari sebuah *object* tidak lebih penting daripada *method*yang menjadi perilakunya.

*Class*, *object*, dan *method* akan kita bahas secara mendalam pada materi kali ini. Kita hanya akan berkenalan terlebih dahulu secara umum sebelum spesifik membahasnya. Sebenarnya, Python ingin memberikan keleluasaan terhadap para developernya untuk tidak perlu mencemaskan tentang tipe atau kelas (class) dari sebuah objek (object), yang lebih penting adalah kemampuan melakukan operasinya (method).

Mari kita ambil contoh, masih ingat dengan fungsi len()? len() merupakan fungsi yang bertujuan untuk menghitung panjang atau banyaknya elemen dari list, set, dan string. Bagaimana dengan tipe data numerik, seperti integer? Perhatikan kode di bawah ini.

|  |
| --- |
| i = 123  print(len(i))  Output:  Traceback (most recent call last):  File "/home/glot/main.py", line 2, in <module>  print(len(i))  TypeError: object of type 'int' has no len() |

Python akan menghasilkan error yang menyatakan bahwa objek integer tidak memiliki len(). Perhatikan lebih baik kalimat error-nya.



Pesan error tersebut menyatakan bahwa objek dengan tipe data "int" tidak memiliki len() yang tersedia, padahal seharusnya method tersebut diharapkan untuk ada. Python secara alami akan memeriksa jika object yang Anda buat memiliki method yang diharapkan atau tidak. Dalam contoh di atas, program menghasilkan error karena object tipe data 'int' tidak memiliki method len().

Baiklah, mari kita mulai perjalanan untuk mempelajari class, object, dan method.

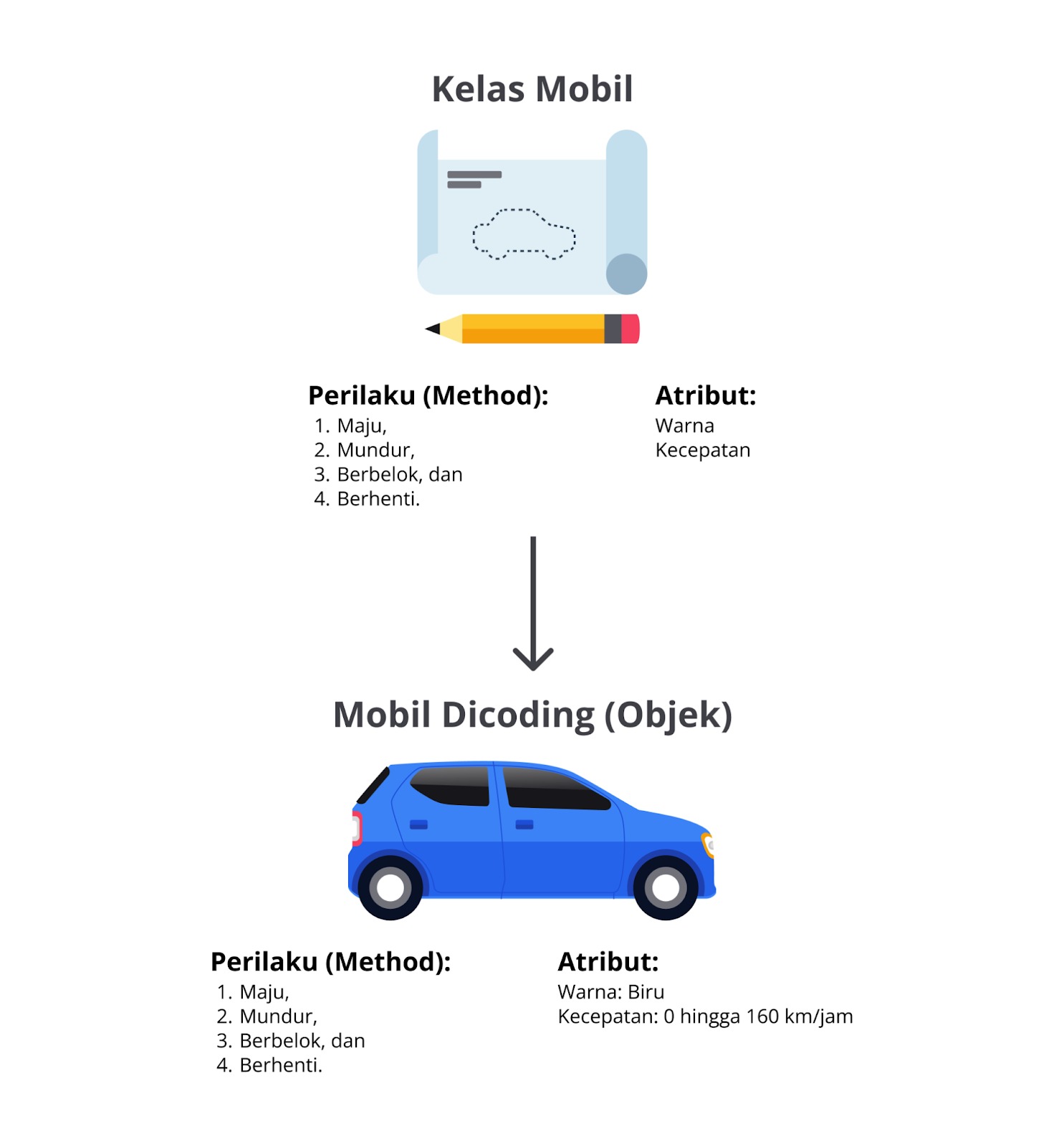
**Class, Object, dan Method**

Sebelum mengenal secara teknis class, object, dan method. Mari kita berandai-andai sejenak untuk memahami konsep **object-oriented programming** (OOP). Bayangkan Anda adalah seorang penggiat mobil, suatu waktu teman Anda yang berasal dari antah-berantah datang menghampiri. Kalian pun mulai berbincang dan dimulai dengan dia yang bertanya satu hal, **"Apa itu mobil?"**.

Anda bisa menjawab, "Mobil adalah jenis kendaraan dengan empat roda yang memiliki kemampuan untuk bergerak maju, mundur, berbelok, dan berhenti. Mobil dapat melaju dengan kecepatan dari 0 hingga 160 km/jam. Mobil juga memiliki variasi warna yang beragam, termasuk merah, hitam, dan warna lainnya."

Berdasarkan ilustrasi tersebut, mobil dapat dikatakan sebagai sebuah kelas (**class**) dan perilaku (**method**) mobil tersebut dapat melaju, mundur, berbelok, dan berhenti. Mobil memiliki **atribut** warna dan warna tersebut bisa beragam, seperti merah, hitam, dan sebagainya serta memiliki **kecepatan** berkisar antara 0 hingga 160 km/jam.

Class dapat diibaratkan sebagai *blueprint* atau cetakan. Dalam contoh percakapan di atas, mobil dapat digambarkan sebagai contoh class atau *blueprint*. Ketika class telah dibuat, Anda dapat membuat sebuah objek baru berdasarkan class tersebut. Objek baru ini memiliki karakteristik, atribut, dan perilaku sama dengan class yang menjadi cetakannya. Anda pun dapat mengubah nilai atribut dari objek tersebut. Perhatikan gambar di bawah ini.



Pada gambar di atas, kita memiliki sebuah kelas bernama Mobil. Kelas ini memiliki method, yaitu bergerak maju, mundur, berbelok, dan berhenti. Dari kelas ini, kita bisa membuat objek baru, misalkan membuat mobil Dicoding.

Objek baru tersebut memiliki unsur method dan atribut sama dengan kelas yang menjadi cetakannya. Bahkan, kita bisa mengubahnya sesuai keinginan. Misalnya pada objek Mobil Dicoding, kita mengubah warnanya menjadi biru. Jika kita tarik ke perandaian lain, ini mirip seperti manusia di seluruh dunia. Kita memiliki teman bernama Budi, Herman, dan Asep yang walaupun nama mereka berbeda, tetapi mereka tetaplah sama-sama manusia seperti kita.

Tidak hanya objek, Anda juga dapat membuat kelas baru untuk mewarisi kelas yang sudah ada.



Terlihat seperti gambar di atas, anggaplah kita memiliki mobil sebagai kelas dasar dengan method maju, mundur, berbelok, dan berhenti. Selain itu, kelas dasar mobil memiliki atribut warna dan kecepatan.

Ketika membuat kelas baru, seperti **Mobil Sport**, kita dapat menggunakan kelas yang sudah ada (mobil sebagai kelas dasar) untuk mewarisi beberapa hal, mulai dari atribut warna, kecepatan hingga beberapa perilakunya, yakni maju, mundur, berbelok, dan berhenti. Namun, kita ingin menambahkan metode baru karena ini adalah **mobil *sport***. Metode baru tersebut adalah turbo yang meningkatkan kecepatan secara signifikan.

Secara umum, konsep OOP dalam pemrograman sangat mirip seperti ilustrasi-ilustrasi di atas. **Object-oriented programming** adalah paradigma pemrograman berorientasi pada pengorganisasian kode menjadi objek-objek yang memiliki atribut dan perilaku (method). Objek merupakan perwujudan dari class dengan anggapan bahwa kelas adalah cetakan yang memungkinkan kita dapat membuat banyak objek berdasarkan cetakan tersebut.

Method adalah perilaku atau tindakan yang dapat dilakukan oleh objek atau kelas. Sebagaimana halnya maju, mundur, berbelok, dan berhenti pada contoh sebelumnya. Atribut adalah variabel yang menjadi identitas dari objek atau kelas, seperti warna dan kecepatan pada contoh sebelumnya.

**Class (Kelas)** Cetakan (blueprint) untuk membuat objek-objek yang memiliki karakteristik dan perilaku serupa.

Mobil; Manusia.

**Object (Objek)** Perwujudan dari kelas.

Mobil Dicoding; Budi, Herman, Asep.

**Perilaku (Method)** Perilaku atau tindakan yang dapat dilakukan oleh objek atau kelas.

Maju, mundur, berbelok, berhenti.

**Atribut** Variabel yang menjadi identitas dari objek atau kelas.

Warna, kecepatan, merek.

**Class**

Pembuatan class dalam Python mirip seperti fungsi, yakni perlu menggunakan keyword untuk bisa membuatnya. Keyword atau kata kunci untuk membuat kelas dalam Python adalah "class". Mari kita buat sebuah kelas bernama mobil.

|  |
| --- |
| class Mobil:  pass |

Pada contoh di atas, kita membuat kelas bernama Mobil. Namun, kelas ini atribut dan method-nya belum didefinisikan sehingga kita masukkan pernyataan "**pass"** supaya tidak menghasilkan error. Ingat bahwa class merupakan blok kode sehingga Anda perlu memperhatikan indentasi untuk membuatnya.

Selanjutnya, mari isi kelas tersebut dengan sebuah atribut. Ingat bahwa atribut adalah variabel yang menjadi identitas dari objek atau kelas.

|  |
| --- |
| class Mobil:  # Atribut  warna = "Merah" |

Pada contoh di atas, kita memberikan variabel warna untuk menjadi atribut atau identitas bawaan dari kelas Mobil. Jika Anda jalankan kode tersebut, tidak akan ada output yang keluar. Hal ini karena kita belum memanggil kelas tersebut dan mendefinisikan kembalian nilai (return).

**Object (Objek)**

Untuk memanggil kelas yang telah dibuat, kita membuat sebuah objek. Berdasarkan KBBI dari kemendikbud, [objek](https://kbbi.kemdikbud.go.id/entri/objek" \t "_blank) merupakan benda, hal, dan sebagainya yang dijadikan sasaran untuk diteliti, diperhatikan, dan sebagainya.

Keterkaitan antara objek dan class sangat erat. Contohnya, jika Anda membuat kelas bernama manusia, objeknya adalah manusia dengan nama yang berbeda. Anda bisa umpamakan kelas adalah bentuk abstrak dari objek, layaknya cetakan atau *blueprint*.

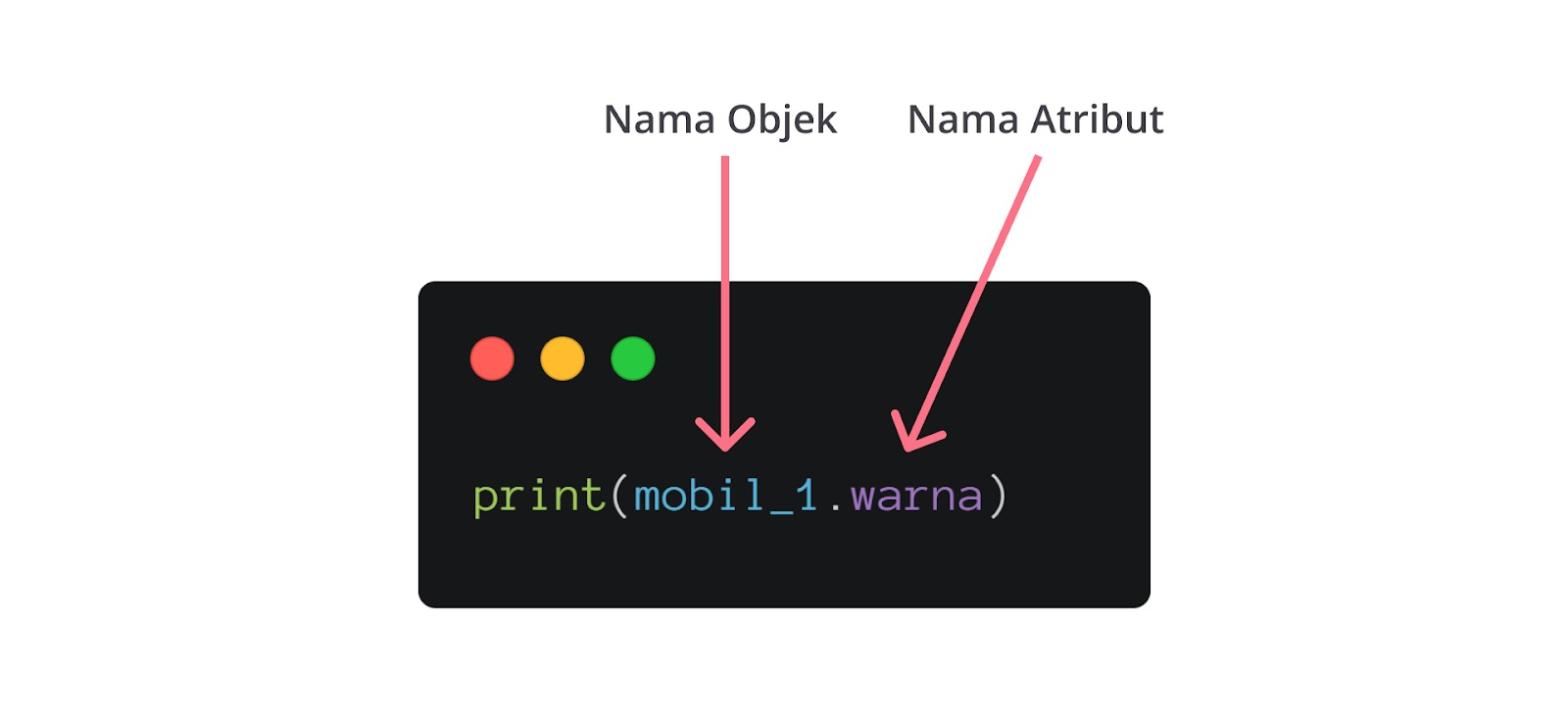
Saat kelas diwujudkan menjadi bentuk yang lebih nyata, proses ini disebut sebagai instansiasi. Itulah sebabnya objek disebut juga sebagai instance atau **instance of the class**.

Pada contoh sebelumnya, kita telah membuat class. Untuk memanggilnya, kita harus mengubahnya menjadi bentuk yang lebih nyata atau bisa dikatakan objek dari kelas tersebut perlu dibuat.

|  |
| --- |
| class Mobil:  # Atribut  warna = "Merah"    mobil\_1 = Mobil() |

Pada contoh di atas, kita memanggil sebuah class dengan menyebutkan namanya diikuti oleh tanda kurung buka dan tutup "()", mirip seperti memanggil fungsi. Selanjutnya pada baris kode "mobil\_1 = Mobil()", kita membuat sebuah objek dari kelas Mobil dan menyimpannya dalam variabel mobil\_1.

Tidak seperti fungsi, variabel dalam contoh di atas adalah objek yang merupakan perwujudan dari kelas yang telah kita buat. Ini berarti variabel tersebut sekarang memiliki atribut yang identik dengan kelas tersebut. Untuk memanggil atribut, kita dapat menyebutkan objek atau instance diikuti dengan nama atributnya.



Dengan membuat objek yang merupakan instance dari kelas, kita pun dapat mengubah atribut tersebut sesuai kebutuhan. Contohnya berikut.

|  |
| --- |
| class Mobil:  # Atribut  warna = 'Merah'  mobil\_1 = Mobil()  mobil\_1.warna = "Biru"  print(mobil\_1.warna) |

Pada contoh di atas, kita mengubah atribut kelas yang awalnya bernilai merah menjadi biru dengan mendeklarasikan ulang nilainya. Perhatikan kode "mobil\_1.warna = 'Biru'", itu merupakan kode yang digunakan untuk mengubah nilai atribut kelas.

**Atribut**

Dalam Python, ada dua jenis atribut kelas yang dapat dibagi, yaitu **atribut kelas**dan **atribut objek atau instance**. Atribut kelas adalah jenis atribut yang secara otomatis terdefinisi dan menjadi bawaan kelas ketika instance dibuat berdasarkan kelas tersebut.

Anda dapat menganggapnya sebagai nilai default atau bawaan dari kelas. Jika Anda membuat beberapa objek berdasarkan kelas yang memiliki jenis atribut ini, setiap objek akan memiliki atribut yang sama dengan nilai yang sama.

Namun, perlu diperhatikan bahwa jenis atribut kelas memiliki kelemahan, yaitu ketika nilai atribut kelas diubah, perubahan tersebut akan memengaruhi semua objek yang dibuat berdasarkan kelas tersebut.

Perhatikan contoh berikut. Asumsikan bahwa kita membuat sebuah kelas bernama "Mobil" dengan atribut "warna". Lalu, dari kelas tersebut kita akan membuat dua objek atau instance.

|  |
| --- |
| class Mobil:  # Atribut kelas  warna = "Merah"  mobil1 = Mobil()  print(mobil1.warna)  mobil2 = Mobil()  print(mobil2.warna)  # Mengubah atribut kelas  Mobil.warna = "Hitam"  print(mobil1.warna)  print(mobil2.warna)  """  Output:  Merah  Merah  Hitam  Hitam  """ |

**Kelemahan ini akan menjadi masalah** jika kita ingin setiap objek memiliki atribut masing-masing yang menjadi ciri khasnya. Sama seperti manusia yang bisa beragam dan mempunyai ciri khas walau dalam satu "*blueprint*" yang sama.

Jenis atribut yang kedua adalah **atribut objek** atau **atribut instance**. Jenis atribut ini memungkinkan setiap instance dari kelas memiliki atribut yang berbeda-beda sesuai dengan keinginan. Namun, untuk membuat **atribut instance** kita perlu menggunakan **class constructor**.

**Class Constructor**

Pembangun kelas atau class constructor adalah sebuah **fungsi khusus**dalam Python yang digunakan untuk menentukan nilai awal atau kondisi awal dari suatu kelas. Dengan fungsi ini, saat kita melakukan proses instansiasi atau pembuatan objek baru, hal pertama yang dilakukan adalah memanggilnya terlebih dahulu.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self):  self.warna = 'Merah' |

Pada contoh di atas, kita membuat fungsi bernama **"\_\_init\_\_"**sebagai fungsi khusus untuk menjadi constructor. Selanjutnya, kita menggunakan parameter **self**,yakni sebuah kata kunci untuk merujuk pada objek yang sedang diproses saat ini.

Ini artinya ketika Anda membuat instance baru bernama "mobil\_1", constructor akan dipanggil pertama kali dan self akan merujuk pada instance atau objek "mobil\_1" tersebut. Begitu pun kalau kita membuat instance baru lainnya bernama "mobil\_2", constructor akan dipanggil pertama kali dan self akan merujuk pada instance "mobil\_2".

Hal ini memungkinkan setiap objek baru tersebut memiliki atribut masing-masing, tidak lagi atribut kelas. Jadi, kita dapat mengubah atribut suatu objek tanpa memengaruhi objek lainnya.

Dengan begitu, **self.warna**yang didefinisikan dalam constructor adalah jenis dari **atribut instance**atau **atribut objek**, yakni atribut yang terkait dengan instance atau objek itu sendiri, bukan kelas.

Mari kita kembali pada contoh kelas mobil dengan atribut warna. Kali ini kita akan mengubah nilai atribut kembali, tetapi perbedaannya adalah atribut tersebut bukan jenis atribut kelas melainkan atribut instance.

|  |
| --- |
| class Mobil:  # Atribut Instance  def \_\_init\_\_(self):  self.warna = 'Merah'    mobil\_1 = Mobil()  mobil\_2 = Mobil()  print(mobil\_1.warna)  print(mobil\_2.warna)  # Mengubah atribut instance  mobil\_1.warna = "Hitam"  # Menampilkan kembali atribut warna  print(mobil\_1.warna)  print(mobil\_2.warna)  Output:  Merah  Merah  Hitam  Merah |

Pada contoh di atas, kita membuat kelas bernama **Mobil**dengan atribut instance adalah **warna** dan nilainya **merah**. Selanjutnya, kita membuat dua instance baru bernama "mobil\_1" dan "mobil\_2". Jika kita cetak atribut kedua objek tersebut ke layar, hasilnya adalah "Merah" untuk kedua atribut tersebut.

Selanjutnya kita ubah atribut instance pada instance "mobil\_1" dengan sintaks **mobil\_1.warna = "Hitam"**. Sintaks ini mengganti atribut objek tersebut dari awalnya merah menjadi hitam.

Untuk mengetahui perbedaannya, kita cetak kembali kedua instance tersebut menggunakan fungsi "print()". Hasilnya atribut instance mobil\_1 berubah menjadi hitam, sedangkan instance mobil\_2 tetap merah.

Jika terlintas dalam benak Anda, mengapa sintaksnya **mobil\_1.warna = "Hitam"** bukan **Mobil.warna = "Hitam"**? Jawabannya dapat dilihat jika Anda mengubah kode di atas dengan kode tersebut. Silakan untuk melakukan *copy-paste* terhadap sintaks yang dimaksud.

Output yang dihasilkan adalah program menampilkan "Merah" semuanya. Artinya atribut pada setiap objeknya tidak terjadi perubahan. Hal ini karena pada kelas yang kita buat tidak memiliki **atribut kelas**, sedangkan sintaks **Mobil.warna = "Hitam"**bertujuan untuk mengubah jenis atribut kelas.

Jika sebelumnya kita hanya menggunakan parameter self saja dalam class constructor, parameter lain pun dapat ditambahkan sesuai kebutuhan.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan    mobil\_1 = Mobil('Merah', 'DicodingCar', 160)  print(mobil\_1.warna)  print(mobil\_1.merek)  print(mobil\_1.kecepatan)  """  Output:  Merah  DicodingCar  160  """ |

Pada contoh di atas, kita membuat kelas yang sama, tetapi ada perbedaan dengan yang sebelumnya. Perbedaannya adalah pada kode di atas, kita membuat parameter tambahan, yaitu "warna, merek, kecepatan". Parameter tambahan ini membuat kita perlu menggunakan argumen ketika memanggil kelas mobil "**Mobil('Merah', 'DicodingCar', 160)**". Jika Anda memanggil kelas tanpa argumen yang disebutkan, program akan menyebabkan error.

Hal penting yang perlu diketahui adalah perbedaan antara kode di atas dengan kode sebelumnya. Sebelumnya, kode hanya menggunakan parameter "self". Lalu, jika kita menggunakan kode seperti berikut.

|  |
| --- |
| def \_\_init\_\_(self):  self.warna = 'Merah' |

Kode di atas berarti kita langsung menentukan nilai default terhadap suatu objek. Itulah sebabnya ketika proses instansiasi, program tidak memunculkan error walaupun kita tidak memberikan argumen apa pun.

|  |
| --- |
| def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan |

Semua atribut yang telah ditetapkan tidak memiliki nilai default. Ketika membuat sebuah objek dari kelas tersebut, kita perlu menambahkan argumen tambahan seperti yang disebutkan, yaitu warna, merek, kecepatan. Inilah sebabnya ketika proses instansiasi atau pembuatan objek, program akan memunculkan error jika tidak memberikan argumen apa pun.

**Method**

Setelah atribut, saatnya membahas method sebagai perilaku atau tindakan yang dapat dilakukan oleh objek atau kelas. Pada pembuatan metode , sebenarnya kita membuat fungsi di dalam kelas itu sendiri. Dengan kata lain, kita menggunakan kata kunci **"def"** atau membuat fungsi sebagai suatu metode. Python membagi method menjadi tiga jenis.

* Metode dari object (*object method*).
* Metode secara statis (*static method*).
* Metode dari class (*class method*).

Dua metode terakhir sangat erat kaitannya dengan konsep dekorator. Kita tidak akan membahasnya lebih detail mengenai dekorator, tetapi secara umum saja.

Dekorator adalah fungsi dalam Python yang mengembalikan fungsi lain, biasanya diawali dengan sintaks "@" di awal.  Contoh sederhana dekorator sebagai berikut.

|  |
| --- |
| def my\_decorator(func):  def wrapper():  print("Sebelum fungsi dieksekusi.")  func()  print("Setelah fungsi dieksekusi.")  return wrapper  # Dekorasi fungsi dengan decorator  @my\_decorator  def say\_hello():  print("Hello, world!")  # Memanggil fungsi yang sudah didekorasi  say\_hello()  """  Output:  Sebelum fungsi dieksekusi.  Hello, world!  Setelah fungsi dieksekusi.  """ |

Penjelasan dari contoh di atas adalah berikut.

1. Pertama, kita mendefinisikan sebuah fungsi bernama **my\_decorator**. Dekorator ini menerima sebuah fungsi **func** sebagai parameternya.
2. Dalam fungsi my\_decorator, kita mendefinisikan fungsi wrapper(). Fungsi wrapper() bertindak sebagai "pembungkus" yang **menambahkan perilaku sebelum dan setelah eksekusi dari fungsi func**.
3. Setelah itu, fungsi  **my\_decorator**mengembalikan (return) fungsi wrapper sebagai hasilnya. Return ini juga menyebabkan fungsi **wrapper**dijalankan.
4. Kemudian, kita mendefinisikan fungsi **say\_hello()**. Fungsi ini akan menjadi target dekorasi.
5. Untuk mendekorasi **say\_hello(),**kita menggunakan simbol "@" diikuti dengan nama dekorator, yaitu **@my\_decorator**sebelum mendefinisikan fungsi say\_hello.
6. Jadi, secara alur, ketika fungsi say\_hello() dipanggil, sebenarnya yang dieksekusi adalah fungsi wrapper() yang menjadi hasil dari dekorasi. Oleh karena itu, pesan "Sebelum fungsi dieksekusi." dicetak terlebih dahulu, kemudian fungsi say\_hello() dieksekusi dan mencetak "Hello, world!", lalu akhirnya, pesan "Setelah fungsi dieksekusi." dicetak.

Dekorator sangat berguna untuk menambahkan fungsionalitas tambahan pada suatu fungsi atau metode tanpa harus mengubah kode asli dari fungsi tersebut. Anda bisa membaca lebih dalam mengenai dekorator pada laman [ini](https://docs.python.org/id/3.8/glossary.html" \l "term-decorator" \t "_blank).

**Metode dari Objek (Object Method)**

Jenis pertama adalah method yang melekat terhadap objek. Ciri dari jenis metode ini adalah **adanya parameter *self*** yang merujuk pada objek saat ini. Perhatikan contoh di bawah ini, asumsikan bahwa kita membuat kelas mobil yang sama seperti sebelumnya. Namun, kita menambahkan metode bernama "tambah\_kecepatan" dan setiap metode ini dipanggil maka kecepatan mobil akan bertambah 10.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan  def tambah\_kecepatan(self):  self.kecepatan += 10  mobil\_1 = Mobil("Merah", "DicodingCar", 160)  print("Sebelum ditambahkan: ")  print(mobil\_1.kecepatan)  mobil\_1.tambah\_kecepatan() # Memanggil metode tambah\_kecepatan.  print("Setelah ditambahkan: ")  print(mobil\_1.kecepatan)  Sebelum ditambahkan: 160 Setelah ditambahkan: 170 |

Pada contoh di atas, kita membuat kelas **Mobil**dan mengimplementasikannya pada **objek bernama "mobil\_1"**. Dalam kelas tersebut, kita memiliki **constructor ('\_\_init\_\_')**yang digunakan untuk menginisialisasi atribut **'warna', 'merek'** dan **'kecepatan'** pada objek yang dibuat (mobil\_1).

Ketika objek 'mobil\_1' dibuat dari kelas Mobil, kita memberikan beberapa argumen yang menjadikannya atribut dari objek tersebut, yakni 'Merah' sebagai warna mobil, 'DicodingCar' sebagai merek, dan 'kecepatan' sebagai kecepatan awal mobil tersebut.

Dalam kelas Mobil yang dibuat, kita menambahkan method berupa fungsi "tambah\_kecepatan" dan digunakan untuk meningkatkan kecepatan mobil. Setiap metode ini dipanggil, kecepatan mobil akan bertambah sebesar 10.

Pada bagian kode berikut lebih tepatnya, kita memanggil metode yang telah dibuat tersebut.

|  |
| --- |
| print("\nSebelum ditambahkan: ")  print(mobil\_1.kecepatan)    mobil\_1.tambah\_kecepatan() # Memanggil metode tambah\_kecepatan.    print("\nSetelah ditambahkan: ")  print(mobil\_1.kecepatan) |

Hasilnya bisa kita lihat pada output dalam program tersebut, bahwa sebelum ditambahkan, kecepatannya adalah 160. Namun setelah itu, kecepatannya bertambah menjadi 170.

Jika menyadarinya, perbedaan ketika Anda memanggil method dan atribut terletak pada penempatan tanda kurung “()”. Ketika memanggil atribut, Anda cukup menyebutkan nama atribut tersebut **tanpa ada**tanda kurung “()”, contohnya “mobil\_1.kecepatan”. Namun untuk memanggil method, Anda **harus menempatkan** tanda kurung “()”setelahnya, contohnya “mobil\_1.tambah\_kecepatan()”.

Selain itu, saat kode di bawah ini dieksekusi,

|  |
| --- |
| mobil\_1.tambah\_kecepatan() |

|  |
| --- |
| Mobil.tambah\_kecepatan(mobil\_1) |

setara dengan kita melakukan kode diatas.

Hal inilah yang dimaksud dengan *self* pada object method karena ketika kita memanggil object method, argumen pertamanya adalah objek dia sendiri (self).

Namun, object method adalah metode yang melekat terhadap suatu objek dan menggunakan parameter *self*. Jadi, kita **tidak bisa**memanggil metode ini langsung melalui kelasnya.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan    def tambah\_kecepatan(self):  self.kecepatan += 10    Mobil.tambah\_kecepatan()  Output:  Traceback (most recent call last):  File "/home/glot/main.py", line 10, in <module>  Mobil.tambah\_kecepatan()  TypeError: tambah\_kecepatan() missing 1 required positional argument: 'self' |

Pada contoh di atas, kita perlu membuat kelas yang sama seperti sebelumnya. Namun kali ini, kita memanggil object method melalui kelasnya secara langsung. Hal ini menyebabkan error karena kita mendefinisikan fungsi tambah\_kecepatan sebagai object method. Artinya, metode ini memerlukan parameter *self* dan merujuk pada objek yang dibuat, sedangkan kita memanggilnya melalui kelas tanpa membuat objek.

Jika ingin membuat metode yang tidak melekat pada objek, kita bisa menggunakan class method atau static method.

**Metode secara Statis (Static Method)**

Static method adalah fungsi atau method pada sebuah kelas yang bersifat statis. Artinya, metode atau fungsi ini bersifat independen dan tidak terikat pada instance kelas. Metode ini dapat dianggap seperti kita membuat fungsi seperti biasa, tetapi didefinisikan dalam kelas sehingga ini menjadi perilaku untuk kelas tersebut. Untuk membuat static method, Anda bisa menambahkan dekorator **@staticmethod**tepat sebelum mendefinisikan fungsi atau metode.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, merek):  self.merek = merek    @staticmethod  def intro\_mobil():  print("Ini adalah metode dari kelas Mobil")    Mobil.intro\_mobil()  mobil\_1 = Mobil("DicodingCar")  mobil\_1.intro\_mobil()  Output:  Ini adalah metode dari kelas Mobil  Ini adalah metode dari kelas Mobil |

Pada contoh di atas, kita membuat sebuah fungsi bernama intro\_mobil() yang menjadi metode atau perilaku dari kelas Mobil dengan diawali oleh dekorator @staticmethod. Fungsi atau metode yang kita buat bernama intro\_mobil dan mencetak pesan "ini adalah metode dari kelas Mobil".

**Metode dari Kelas (Class Method)**

Metode terakhir adalah class method yang termasuk jenis metode cukup spesial dalam Python. Jika object method identik dengan parameter *self* yang merujuk pada objek, **class method**juga memerlukan sebuah parameter yang merujuk pada kelas. Mari buat contoh yang sama dengan sebelumnya, tetapi kali ini menggunakan class method.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, merek):  self.merek = merek  @classmethod  def intro\_mobil(cls):  print("Ini adalah metode dari kelas Mobil")    Mobil.intro\_mobil()  mobil\_1 = Mobil("DicodingCar")  mobil\_1.intro\_mobil()  """  Output:  Ini adalah metode dari kelas Mobil  Ini adalah metode dari kelas Mobil  """ |

Pada contoh di atas, kita membuat program yang sama, tetapi ada sedikit perbedaan, yakni dekorator @classmethod digunakan. Perhatikan baik-baik pada bagian kode berikut.

|  |
| --- |
| @classmethod  def intro\_mobil(cls):  print("Ini adalah metode dari kelas Mobil") |

Pada bagian fungsi intro\_mobil, kita menambahkan parameter **cls**. Ini adalah parameter wajib dalam menggunakan dekorator @classmethod.

**Catatan:**  
Penamaan cls merupakan kesepakatan bersama dari programmer Python untuk memudahkan pembacaan kode. Anda dapat mengganti namanya, tidak harus cls.

Mengapa demikian? Sebab, ketika menggunakan class method, kita akan menambahkan argumen tambahan pada posisi pertama berupa kelas itu sendiri. Perhatikan kode berikut.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, merek):  self.merek = merek  @classmethod  def intro\_mobil(\*args):  print(args)    Mobil.intro\_mobil()  mobil\_1 = Mobil("DicodingCar")  mobil\_1.intro\_mobil()  """  Output:  (<class '\_\_main\_\_.Mobil'>,)  (<class '\_\_main\_\_.Mobil'>,)  """ |

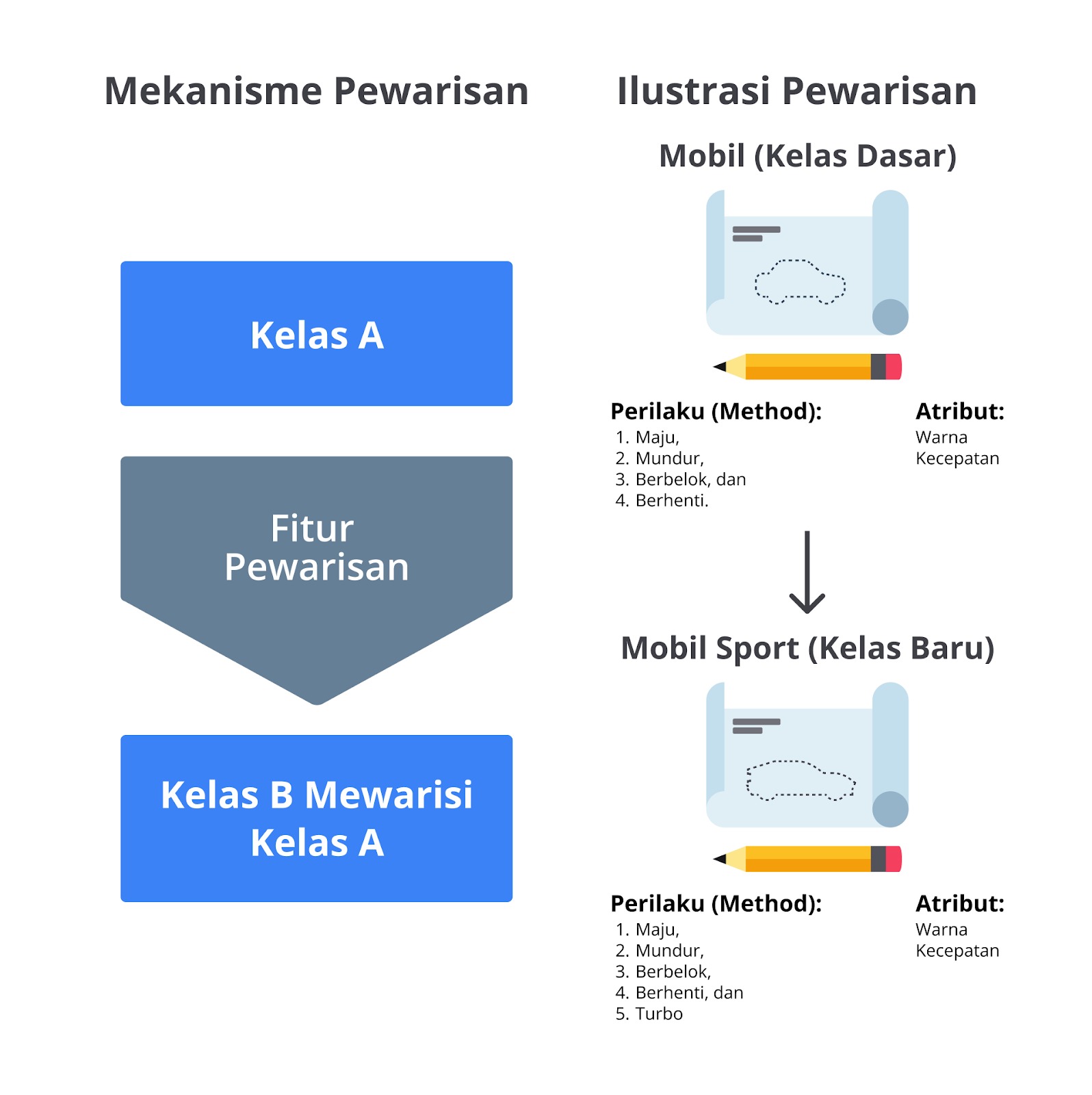
Pada contoh kode di atas, fungsi intro\_mobil() menggunakan variadic positional parameter, yakni \*args untuk melihat argumen yang diterima oleh fungsi tersebut. Perhatikan lebih baik output-nya, kode di atas menerima sebuah parameter, yakni kelas Mobil walaupun ketika pemanggilan fungsi intro\_mobil() kita tidak memberikan argumen apa pun.

|  |
| --- |
| Mobil.intro\_mobil()  mobil\_1 = Mobil("DicodingCar")  mobil\_1.intro\_mobil() |

**Inheritance (Pewarisan)**

Sebagaimana ilustrasi awal, kita dapat membuat sebuah kelas baru dengan menggunakan kelas induk yang sudah ada. Konsep ini disebut dengan 'inheritance' atau dalam bahasa Indonesia artinya pewarisan.

**Mekanisme Pewarisan**



Untuk melakukan pewarisan, anggap kita memiliki "kelas A" sebagai induk atau kelas dasar. Dari kelas A tersebut kita membuat kelas baru bernama "kelas B" sebagai kelas turunan dari kelas yang didapatkan (kelas A). Ketika kelas B mewarisi kelas A, secara otomatis kelas ini memiliki fitur-fitur yang dimiliki oleh kelas A tersebut, dalam hal ini atribut-atribut dan metode-metode.

Sebagaimana halnya pada ilustrasi di sampingnya, Anda mempunyai **kelas Mobil**sebagai kelas dasar atau induk. Lalu, kita membuat **kelas Sport**sebagai turunan dari kelas mobil yang sudah ada. Selanjutnya, kita bisa memiliki perilaku dan atribut yang sama dengan kelas sebelumnya. Bahkan kita bisa menambahkan hal baru, seperti perilaku turbo pada kelas mobil sport.

Hal yang perlu diperhatikan, jika kelas B memiliki nama metode yang sama dengan kelas A, metode tersebut akan menimpa metode yang diwariskan oleh kelas A.

Mari lihat contoh pewarisan di bawah ini, asumsikan bahwa kita perlu membuat kelas Mobil sebagai kelas dasar. Dari kelas Mobil ini, kita akan membuat kelas Mobil baru bernama Mobil Sport dengan fitur yang sama seperti kelas dasarnya. Namun, ada tambahan fitur baru pada kelas tersebut, yakni turbo untuk meningkatkan kecepatan mobil hingga 50 km/jam.

Mari mulai dengan kelas dasar Mobil.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan    def tambah\_kecepatan(self):  self.kecepatan += 10  mobil\_1 = Mobil("Merah", "DicodingCar", 160)  print(mobil\_1.kecepatan)  """  Output:  160  """ |

Pada contoh di atas, kita membuat sebuah kelas bernama Mobil dengan atributnya adalah warna, merek, dan kecepatan. Kelas ini juga memiliki metode, yaitu tambah\_kecepatan untuk meningkatkan kecepatan mobil sebesar 10. Anggap bahwa kode ini tidak boleh diubah sama sekali.

Kita akan membuat sebuah kelas baru bernama "MobilSport" yang mewarisi atribut dan metode kelas Mobil dasar. Lalu, kita menambahkan metode baru, yaitu turbo untuk meningkatkan kecepatan sebesar 50.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan    def tambah\_kecepatan(self):  self.kecepatan += 10  class MobilSport(Mobil):  def turbo(self):  self.kecepatan += 50  # Kelas Mobil Dasar  mobil\_1 = Mobil("Merah", "DicodingCar", 160)  print(mobil\_1.kecepatan)  # Kelas Mobil Sport  mobil\_sport\_1 = MobilSport("Hitam", "DicodingSportCar", 160)  print(mobil\_sport\_1.kecepatan)  mobil\_sport\_1.turbo()  print(mobil\_sport\_1.kecepatan)  """  Output:  160  160  210  """ |

Pada contoh di atas, kita membuat kelas baru bernama MobilSport yang mewarisi kelas sebelumnya, yaitu kelas Mobil. Mari fokus terlebih dahulu pada bagian kode berikut.

|  |
| --- |
| class MobilSport(Mobil):  def turbo(self):  self.kecepatan += 50 |

Pada bagian kode tersebut, kita mendefinisikan kelas MobilSport dengan menambahkan parameter tambahan, yaitu kelas Mobil. Dengan demikian, kelas MobilSport akan mewarisi seluruh fitur dari kelas Mobil, seperti atribut warna, merek, kecepatan, dan sebagainya.

Dalam kelas MobilSport, kita membuat object method baru, yaitu turbo untuk meningkatkan kecepatan mobil. Perhatikan bahwa dalam kelas tersebut, kita tidak perlu mendefinisikan kembali atribut sehingga parameter *self* akan merujuk pada atribut bawaan, yakni kelas Mobil.

Selanjutnya, pada bagian kode berikut.

|  |
| --- |
| # Kelas Mobil Sport  mobil\_sport\_1 = MobilSport("Hitam", "DicodingSportCar", 160)  print(mobil\_sport\_1.kecepatan)  mobil\_sport\_1.turbo()  print(mobil\_sport\_1.kecepatan) |

Pada contoh di atas, kita memanggil kelas baru dan membuat objek bernama "mobil\_sport\_1". Perhatikan bahwa kita juga perlu menambahkan argumen sesuai atribut yang didefinisikan.

Perintah "print()" pertama akan mencetak kecepatan mobil\_sport\_1 saat ini, yaitu 160, dan itu adalah nilai bawaan dari kelas mobil dasar. Lalu, kita memanggil metode baru yang telah dibuat, yaitu "**mobil\_sport\_1.turbo()**". Hal ini menyebabkan kita memanggil atribut kecepatan yang bertambah dari 160 menjadi 210 (bertambah 50).

Dengan melakukan pewarisan, Anda dengan mudah bisa menambahkan (*extend*) kemampuan dari suatu kelas dengan fitur yang dibuat sendiri.

**Override**

Selanjutnya, seperti yang dijelaskan di awal, ketika kita membuat metode baru di kelas turunan (kelas baru) dengan nama yang sama seperti metode di kelas induk, itu akan menyebabkan metode baru menimpa (*override*) metode dari kelas induk.

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan    def tambah\_kecepatan(self): # tambah\_kecepatan  self.kecepatan += 10  class MobilSport(Mobil):  def turbo(self):  self.kecepatan += 50    def tambah\_kecepatan(self): # tambah\_kecepatan  self.kecepatan += 20  # Kelas Mobil Sport  mobil\_sport\_1 = MobilSport("Hitam", "DicodingSportCar", 160)  print(mobil\_sport\_1.kecepatan)  mobil\_sport\_1.tambah\_kecepatan() # Memanggil metode baru tambah kecepatan()  print(mobil\_sport\_1.kecepatan)  """  Output:  160  180  """ |

Pada contoh di atas, kita menambahkan metode baru bernama **tambah\_kecepatan**. Metode ini juga ada di kelas mobil dasar. Namun, kita melakukan perbedaan pada metode baru ini berupa penambahan kecepatan yang awalnya sebesar 10 di kelas induk, menjadi sebesar 20 di kelas baru. Hasilnya, dapat kita lihat bahwa kecepatan kini bertambah 20 setiap kita memanggil metode **tambah\_kecepatan()**.

Namun, perlu dipahami bahwa menimpa bukan berarti mengubah metode dari kelas induk. Hal ini karena metode dari kelas baru tersebut merupakan hasil dari pewarisan sehingga tidak akan mengubah metode dari kelas induk. Silakan tambahkan kode berikut ke program di atas pada bagian akhir kode (sebelum komentar output).

|  |
| --- |
| # Kelas Mobil Dasar  mobil\_1 = Mobil("Merah", "DicodingCar", 160)  print(mobil\_1.kecepatan)  mobil\_1.tambah\_kecepatan()  print(mobil\_1.kecepatan) |

Anda akan melihat bahwa ketika kita membuat objek dari kelas dasar, lalu memanggil metode dari kelas dasar tersebut, program tetap menambah kecepatan sebesar 10 walaupun kita sudah melakukan overriding (menimpa) di kelas baru.

Saat Anda menjalankan program, Python akan mencari nama metode dengan nama yang sesuai di kelas baru terlebih dahulu. Jika tidak ada, nama akan dicari di kelas induk.

**Super**

Lantas, bagaimana cara untuk kita ingin menggunakan metode atau atribut dari kelas induk, tetapi tidak ingin menuliskan ulang semua kode? Ini adalah tujuan dari adanya **super** dalam konsep OOP. Nama super sebenarnya merujuk pada kelas induk yang disebut juga sebagai **super class**. Kita bisa memanfaatkan konsep ini untuk menghindari kode berulang dan memanfaatkan fungsi yang sudah ada pada kelas induk (super class).

Mari kita lihat contohnya, kita akan menggunakan program yang sama seperti pada materi overriding. Namun, alih-alih menambah kecepatan, mari kita tambahkan teks baru berupa "Kecepatan Anda meningkat! Hati-Hati!".

|  |
| --- |
| class Mobil:  def \_\_init\_\_(self, warna, merek, kecepatan):  self.warna = warna  self.merek = merek  self.kecepatan = kecepatan    def tambah\_kecepatan(self):  self.kecepatan += 10  class MobilSport(Mobil):  def turbo(self):  self.kecepatan += 50    def tambah\_kecepatan(self):  super().tambah\_kecepatan()  print("Kecepatan Anda meningkat! Hati-Hati!")  # Kelas Mobil Sport  mobil\_sport\_1 = MobilSport("Hitam", "DicodingSportCar", 160)  mobil\_sport\_1.tambah\_kecepatan() # Memanggil metode baru tambah kecepatan()  print(mobil\_sport\_1.kecepatan)  Output:  Kecepatan Anda meningkat! Hati-hati!  170 |

Pada contoh di atas, kita membuat kelas MobilSport sebagai kelas turunan dengan metode tambahan, yaitu tambah\_kecepatan().

|  |
| --- |
| def tambah\_kecepatan(self):  super().tambah\_kecepatan()  print("Kecepatan Anda meningkat! Hati-Hati!") |

Pada metode ini, kita menggunakan "super()" untuk mengambil metode tambah\_kecepatan yang berasal dari super class atau induknya, yaitu kelas Mobil. Dengan begitu, program akan menjalankan metode tersebut dan di bawahnya kita menambahkan teks baru sesuai kebutuhan pada kelas turunan berupa "Kecepatan Anda meningkat! Hati-hati!".