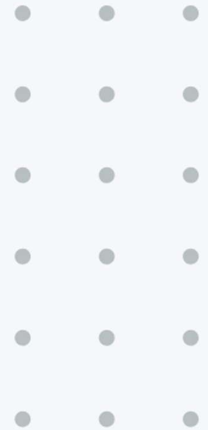
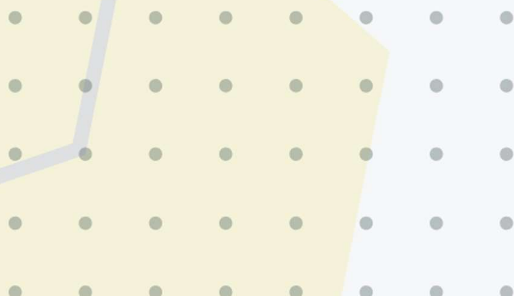
TEKNIK INFORMARTIKA



MODUL PRAKTIKUM

PEMROGRAMAN BERBASIS OBJEK

Pertemuan 2
Object Oriented Programming



Halo semuanya, dalam pertemuan kali ini kita akan membahas tentang OOP (Object Oriented Programming) di dalam Python. OOP adalah salah satu dari banyak konsep di dalam program komputer di mana beberapa konsep digunakan yaitu:

1. Encapsulation
2. Inheritance
3. Abstraction (Pertemuan Selanjutnya)
4. Polymorphism (Pertemuan Selanjutnya)

CLASS

Classes adalah konsep utama dalam Object Oriented Programming di banyak bahasa pemrograman (Javar, C++, dll). Classes membuat program menjadi bentuk bagian - bagian kecil, sehingga bisa mengurangi kemungkinan terjadinya error. Di dalam OOP, ada beberapa istilah penting yang perlu diketahui:

1. Kelas (Class)**:** Kelas itu kayak cetakan atau blueprint yang nentuin gimana bentuk dan kelakuan suatu objek. Di dalamnya ada data (atribut/properti) dan juga fungsi (metode) yang bisa dipakai buat mengolah data itu.
2. Objek (Object)**:** Objek adalah hasil nyata dari sebuah kelas, kayak benda hasil cetakan. Dia punya data sendiri dan bisa ngelakuin hal-hal yang udah ditentuin di dalam kelasnya.
3. Metode (Method)**:** Metode itu fungsi yang ada di dalam kelas. Dia dipakai buat ngasih perilaku ke objek, kayak mengolah data atau berinteraksi sama objek lain.
4. Properti (Property)**:** Properti (atau atribut) adalah data yang disimpan dalam objek. Ini yang bikin setiap objek punya karakteristik unik sesuai dengan kelasnya.

```python
# A simple example classes
class MyClass(): # Class itself
    myProperty = "Sebastian"  # Class's Property

    def myMethod(self): # Class's Method
        print(f"Aloha {self.myProperty}\n")

myObject = MyClass() # Class's Objects
myObject.myMethod() # Aloha Sebastian
```

Kata kunci self di Python itu dipakai di dalam metode kelas buat nunjukin kalau kita lagi akses atribut atau metode dari objek (instance) itu sendiri. Kalau nggak pakai self, Python bakal nyangka kita lagi pakai variabel biasa (lokal/global), bukan atribut yang ada di dalam objek.

A. **Kenapa harus pakai self?**

1. **Biar bisa akses atribut dari objeknya**
   Kalau kita pakai self.myProperty, itu artinya atribut itu milik objeknya, jadi tiap objek bisa punya nilai yang beda meskipun pakai atribut yang sama.
2. **Biar nggak ketuker sama variabel lokal**
   Tanpa self, Python bakal mikir kita pakai variabel biasa di dalam metode, bukan atribut dari objek yang udah dibuat.

```
1  myProperty = "Ceb"
2
3  # A simple example classes
4  class MyClass(): # Class itself
5     myProperty = "Sebastian"  # Class's Property
6
7     def myMethod(self): # Class's Method
8        print(f"Aloha {myProperty}\n")
9
10 myObject = MyClass() # Class's Objects
11 myObject.myMethod() # Aloha Ceb
```

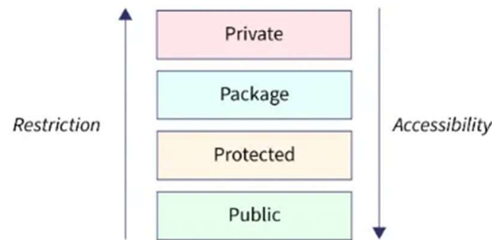
   Classes bisa dipelajari lebih lanjut di
   https://docs.python.org/3/tutorial/classes.htm

B. Access Modifier dalam Python

Access Modifier di Python itu ngatur "siapa aja yang bisa lihat dan pakai" atribut atau metode dari sebuah kelas. Ini berguna buat nentuin apakah objek bisa mengakses, mengubah, atau mewarisi anggota tertentu dari kelas. Secara default, Python punya **tiga jenis access modifier:**

1. Public **(Publik)**
   - Semua anggota kelas (atribut dan metode) bersifat **publik** secara default.
   - Artinya, bisa diakses dari mana aja, baik dari dalam kelas maupun dari luar.
2. Private **(Pribadi)**
   - Hanya bisa diakses di dalam kelas itu sendiri.
   - Dideklarasikan dengan **double underscore (__)** di depan nama atribut/metode.
   - Contoh: __myPrivateVar
3. Protected **(Dilindungi)**
   - Hanya bisa diakses oleh **kelas itu sendiri dan turunannya (subclass)**.
   - Dideklarasikan dengan **single underscore (_)** di depan nama atribut/metode.
   - Contoh: _myProtectedVar

Jadi, kalau kita mau bikin variabel/metode yang **cuma bisa dipakai di dalam kelas aja**, pakai __ (private). Kalau mau bisa dipakai di kelas itu dan turunannya (inherit), pakai _ (protected). Kalau mau bebas diakses dari mana aja, biarkan aja tanpa underscore (public).




```python
# Some example classes about access modifier
class MyClass():
    my_public_property = "Hi, Public"
    _my_protected_property = "Hi, Protected"
    __my_private_property = "Hi, Private"

myObject = MyClass()
print(myObject.my_public_property)
print(myObject._my_protected_property)
print(myObject.__my_private_property)
```

C. Cara Mengakses/Mengubah Private Member di Luar Kelas

Kalau atribut atau metode bersifat **private** (diawali __), kita nggak bisa langsung akses dari luar kelas. Tapi, kita bisa pakai **getter** dan **setter** untuk membaca atau mengubah nilainya dengan cara yang lebih aman.

- **Getter**: Digunakan untuk mengambil (membaca) nilai dari atribut private.
- **Setter**: Digunakan untuk mengubah nilai atribut private dengan validasi tertentu.

```python
class BankAccount():
    def __init__(self, accountNumber, balance):
        if (balance < 0) : # Conditional check to handle the negative input
            print("Please Give the Correct Number of Balance")
            exit()

        self.__accountNumber = accountNumber
        self.__balance = balance
        self.show_info() # Calling the show_info() when first initializationo

    def show_info(self): # Getter Function
        print(f"Halo, Rekening Anda: {self.__accountNumber}, dan jumlah saldo Anda: Rp. {self.__balance}\n")

    def deposit(self, amount): # Setter Function
        self.__balance += amount

    def withdraw(self, amount): # Setter Function
        if (self.__balance >= amount):
            self.__balance -= amount
            print(f"Terima kasih atas penarikan saldo sebesar Rp. {amount}")
        else:
            print("Saldo tidak cukup :(")

# Creating an instance of BankAccount
myAccount = BankAccount("123456789", 1000);
myAccount.deposit(500);
myAccount.withdraw(200);
```

D. Encapsulation

Encapsulation itu konsep di OOP yang **ngumpulin data (atribut) dan fungsi (metode) dalam satu unit, yaitu kelas**. Dengan cara ini, data bisa lebih aman karena cuma bisa diakses lewat metode tertentu (misalnya, lewat getter dan setter).

Kenapa penting?

- Mencegah modifikasi langsung terhadap atribut yang seharusnya tidak diubah sembarangan.
- Memastikan data hanya bisa diubah dengan cara yang benar (misalnya, saldo nggak boleh negatif).
- Memudahkan pengelolaan kode karena atribut tersembunyi dan hanya bisa diakses lewat metode tertentu.

Contoh Encapsulation dalam Kelas BankAccount

```
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Berhasil menambah saldo: {amount}")
        else:
            print("Jumlah deposit harus lebih dari 0!")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Berhasil menarik saldo: {amount}")
        else:
            print("Saldo tidak mencukupi atau jumlah tidak valid!")

    def get_balance(self):
        return self.__balance  # Getter untuk membaca saldo

# Contoh penggunaan
account = BankAccount("Alice", 5000)
account.deposit(2000)  # Berhasil menambah saldo: 2000
account.withdraw(3000)  # Berhasil menarik saldo: 3000
print(account.get_balance())  # Output: 4000
```

Dalam contoh ini, kita **menyembunyikan atribut** __balance supaya nggak bisa diakses langsung dari luar. Akses saldo hanya bisa dilakukan lewat metode get_balance(), deposit(), dan withdraw().

Berikut adlaah artikel tambahan untuk mempelajari tentang Encapsulation, klik di https://medium.com/@kateolenya/encapsulation-in-python-3-1a93fd8fa9cd

E. Inheritance (Pewarisan) dalam Python

Inheritance (pewarisan) itu konsep di OOP yang memungkinkan kita bikin **kelas baru (subclass/anak)** yang **ngebawa atribut dan metode dari kelas lama (superclass/induk)**. Dengan inheritance, kita bisa **pakai ulang kode tanpa harus nulis ulang**, jadi lebih efisien dan gampang dikembangkan.
Selain itu, kita juga bisa **nambahin atau ngubah fitur** dari kelas induk di subclass tanpa harus ngotak-atik kode aslinya.

```
1  class Animal():
2      def __init__(self, name, age, species):
3        self.name = name
4        self.age = age
5        self.species = species
6
7      def eat(self, food):
8          print(f"{self.name} is eating {food}.")
9
10     def sleep(self, hours):
11         print(f"{self.name} is sleeping for {hours} hours.")
12
13     def make_sound(self): # Some Basic Concept of Abstraction
14         raise NotImplementedError("Subclasses must implement this method.")
```

Kita bisa bikin **kelas anak (subclass)** seperti Dog atau Cat yang ngewarisin atribut dan metode dari kelas induk Animal. Ini yang bikin inheritance jadi berguna banget, karena kita **nggak perlu nulis ulang kode yang sama**, tapi tetap bisa **menyesuaikan atau nambah fitur khusus** di tiap subclass.

```
1  class Dog(Animal):
2      def __init__(self, name, age, breed):
3          super().__init__(name, age, species = "Dog")
4          self.breed = breed
5
6      def make_sound(self):
7          print("Woof! Woof!")
8
9  animal = Dog("Ceb", 13, "Alaskan")
10 animal.eat("Meat")
11 animal.make_sound()
```

```
1  class Cat(Animal):
2      def __init__(self, name, age, breed, eye_color):
3          super().__init__(name, age, species = "Cat")
4          self.breed = breed
5          self.eye_color = eye_color
6
7      def make_sound(self):
8          print("Meow...")
```

F. Penjelasan Lebih Lanjut tentang Inheritance & Overriding

1. **Mewarisi Properti & Metode dari Kelas Induk**
   - Saat kita bikin subclass Dog dan Cat, kita **memasukkan kelas** Animal **dalam deklarasi** supaya bisa mewarisi semua atribut & metode dari Animal.
   - Contohnya, Dog dan Cat **langsung punya** atribut name dan metode make_sound() tanpa harus menulis ulang.
2. **Pakai** super() **buat Manggil Kelas Induk**
   - Python punya fungsi super(), yang memungkinkan subclass **manggil metode dari kelas induk**.
   - Ini berguna kalau kita mau **nambahin fitur baru tapi tetap pakai metode induknya**.
3. **Overriding Metode (**make_sound()**)**
   - Dog dan Cat **mengganti (override)** metode make_sound() supaya masing-masing punya suara sendiri.
   - Overriding memungkinkan kita bikin **perilaku spesifik di setiap subclass**, menggantikan metode bawaan dari superclass.

```
1  # Kelas Induk (Super Class)
2  class Animal:
3      def __init__(self, name):
4          self.name = name
5
6      def make_sound(self):  # Metode bawaan di kelas induk
7          return "Hewan ini mengeluarkan suara."
8
9  # Kelas Anak (Subclass) yang Override make_sound()
10 class Dog(Animal):
11     def make_sound(self):  # Override metode dari Animal
12         return f"{self.name} menggonggong: 'Guk Guk!'"
13
14 class Cat(Animal):
15     def make_sound(self):  # Override metode dari Animal
16         return f"{self.name} mengeong: 'Meong!'"
17
18 # Membuat objek dari subclass
19 dog1 = Dog("Buddy")
20 cat1 = Cat("Whiskers")
21
22 # Memanggil metode yang sudah dioverride
23 print(dog1.make_sound())  # Output: Buddy menggonggong: 'Guk Guk!'
24 print(cat1.make_sound())  # Output: Whiskers mengeong: 'Meong!'
```

Di contoh sebelumnya, make_sound() punya implementasi default. Tapi kalau kita mau memastikan bahwa **semua subclass HARUS punya metode ini**, kita bisa bikin **metode abstrak** di kelas induk.
Metode abstrak adalah **metode yang dideklarasikan tapi belum diimplementasikan**, jadi subclass harus mendefinisikannya sendiri.
Materi inheritance bisa dipelajari lebih lanjut di https://www.w3schools.com/python/python_inheritance.asp.