TUGAS PRAKTIKUM

PEMROGRAMAN BERORIENTASI OBJEK

Minggu 5

Disusun Oleh :

Bagas Satrio 121140077

PROGRAM STUDI TEKNIK INFORMATIKA

JURUSAN TEKNIK ELEKTRO, INFORMATIKA DAN SISTEM FISIS

INSTITUT TEKNOLOGI SUMATERA

2023

DAFTAR ISI

BAB I

OBJEK DAN KELAS DALAM PYTHON

(KONSTRUKTOR, SETTER, DAN GETTER)

A. KELAS

kelas (class) adalah sebuah blueprint atau template untuk menciptakan objek. Kelas berisi atribut dan metode yang mendefinisikan perilaku dan sifat dari objek yang dibuat dari kelas tersebut. Sebagai contoh, jika kita ingin membuat sebuah kelas untuk merepresentasikan mahasiswa, kita bisa mendefinisikan atribut seperti nama, usia, dan jurusan, serta metode seperti mengambil atau mengubah data mahasiswa.

Berikut adalah contoh implementasi kelas pada python untuk membuat kelas mahasiswa:

```
1. class Mahasiswa:
2.
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         self.nama = nama
5.         self.nim = nim
6.         self.pbo = pbo
7.         self.jumsks = jumsks
8.     def tampil_data(self):
9.         print ("Nama = ", self.nama)
10.        print ("NIM = ", self.nim)
11.        print ("Kelas PBO = ", self.pbo)
12.        print ("Jumlah SKS = ", self.jumsks)
13.
14.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
15.data.tampil_data()
```

Pada contoh diatas terdapat sebuah kelas bernama **“Mahasiswa”** dengan atribut nama, usia, dan jurusan, serta metode tampil_data yang akan mencetak informasi mahasiswa ke layar. Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama : Bagas Satrio
Nim : 121140077
Kelas PBO : RB
Jumlah SKS : 22
```

Sebuah kelas terdapat variabel (atribut/properti) dan fungsi (method).

1. Atribut

Atribut pada kelas Python adalah variabel yang didefinisikan di dalam kelas dan digunakan untuk menyimpan data yang terkait dengan objek yang dibuat dari kelas tersebut. Setiap objek yang dibuat dari kelas tersebut memiliki salinan atribut yang sama dengan nilai yang berbeda-beda. Atribut dapat didefinisikan baik di dalam metode kelas maupun di luar metode kelas. Atribut yang didefinisikan di luar metode kelas disebut dengan atribut kelas (class attribute) dan nilainya akan sama untuk setiap objek yang dibuat dari kelas tersebut. Sedangkan atribut yang didefinisikan di dalam metode kelas disebut dengan atribut objek (instance attribute) dan nilainya dapat berbeda-beda untuk setiap objek yang dibuat dari kelas tersebut.

Berikut adalah contoh implementasi Atribut pada python untuk membuat kelas mahasiswa:

```
1. class Mahasiswa:
2.
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         self.nama = nama
5.         self.nim = nim
6.         self.pbo = pbo
7.         self.jumsks = jumsks
8.     def tampil_data(self):
9.         print ("Nama = ", self.nama)
10.        print ("NIM = ", self.nim)
11.        print ("Kelas PBO = ", self.pbo)
12.        print ("Jumlah SKS = ", self.jumsks)
13.
14.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
15.data.tampil_data()
```

Pada contoh diatas, terdapat atribut objek **“nama”**, **“nim”**, **“pbo”**, dan **“jumsks”** yang masing-masing memiliki nilai yang berbeda-beda untuk setiap objek yang dibuat dari kelas **“Mahasiswa”**. Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama : Bagas Satrio
Nim : 121140077
Kelas PBO : RB
Jumlah SKS : 22
```

2. Method

Method pada Python adalah fungsi yang didefinisikan di dalam kelas dan digunakan untuk memanipulasi objek yang dibuat dari kelas tersebut. Method biasanya digunakan untuk mengubah nilai atribut objek, melakukan operasi pada objek, atau mengembalikan nilai yang dihasilkan dari objek.

```
1. class Mahasiswa:
2.
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         self.nama = nama
5.         self.nim = nim
6.         self.pbo = pbo
7.         self.jumsks = jumsks
8.     def tampil_data(self):
9.         print ("Nama = ", self.nama)
10.        print ("NIM = ", self.nim)
11.        print ("Kelas PBO = ", self.pbo)
12.        print ("Jumlah SKS = ", self.jumsks)
13.
14.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
15.data.tampil_data()
```

Pada contoh diatas, terdapat metode **"tampil_data"** yang akan mencetak informasi mahasiswa ke layar. Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama : Bagas Satrio
Nim : 121140077
Kelas PBO : RB
Jumlah SKS : 22
```

B. OBJEK

Objek pada Python adalah suatu instance atau perwujudan dari suatu kelas atau tipe data tertentu. Objek terdiri dari data atau nilai-nilai yang disimpan di dalamnya (disebut juga atribut) dan fungsi-fungsi atau metode yang dapat diterapkan pada objek tersebut. Setiap objek memiliki tipe data tertentu yang menentukan jenis nilai yang dapat disimpan di dalamnya dan fungsi atau metode yang dapat diterapkan pada objek tersebut. Contohnya, tipe data string memungkinkan penyimpanan teks atau karakter dan memiliki metode

seperti upper() untuk mengubah teks menjadi huruf besar atau split() untuk memecah teks menjadi bagian-bagian yang lebih kecil berdasarkan separator tertentu.

Contoh : Diibaratkan kita memiliki sebuah kelas bernama "**Mahasiswa**", maka untuk membuat objek bernama "**data**" cukup seperti ini :

```
data = Mahasiswa()
```

C. MAGIC METHOD

Magic method atau metode ajaib pada Python adalah metode khusus yang memiliki nama yang diawali dan diakhiri dengan double underscore (dunder), seperti __init__, __str__, __add__, dan lain sebagainya. Metode ini digunakan untuk memberikan perilaku khusus pada objek saat suatu operasi atau aksi tertentu dilakukan pada objek tersebut. Magic method biasanya digunakan pada kelas, dan memberikan perilaku khusus pada kelas tersebut. Contohnya, metode __init__ digunakan untuk menginisialisasi objek ketika objek dibuat, metode __str__ digunakan untuk mengubah objek menjadi string, dan metode __add__ digunakan untuk menentukan perilaku penjumlahan (+) pada objek.

Contoh implementasi magic method __init__() pada python :

```
1. class Mahasiswa:
2.
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         self.nama = nama
5.         self.nim = nim
6.         self.pbo = pbo
7.         self.jumsks = jumsks
8.     def tampil_data(self):
9.         print ("Nama = ", self.nama)
10.        print ("NIM = ", self.nim)
11.        print ("Kelas PBO = ", self.pbo)
12.        print ("Jumlah SKS = ", self.jumsks)
13.
14.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
15.data.tampil_data()
```

D. KONSTRUKTOR

Konstruktor pada Python adalah metode khusus pada sebuah kelas yang dipanggil saat objek dari kelas tersebut dibuat. Nama konstruktor pada Python adalah __init__(), dan metode ini dapat digunakan untuk menginisialisasi nilai awal dari atribut-atribut pada objek. Konstruktor pada Python sering digunakan untuk memberikan nilai awal pada atribut-atribut pada objek ketika objek dibuat dan menjalankan kode atau operasi tertentu saat objek dibuat.

Contoh implementasi Konstruktor pada python :

```
1. class Mahasiswa:
2.     #Constructor
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         self.nama = nama
5.         self.nim = nim
6.         self.pbo = pbo
7.         self.jumsks = jumsks
8.
9. data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
10.
11.print(f'Nama : {data.nama}')
12.print(f'Nim : {data.nim}')
13.print(f'Kelas PBO : {data.pbo}')
14.print(f'Jumlah SKS : {data.jumsks}')
```

E. DESTRUKTOR

Destruktor pada Python adalah metode khusus pada sebuah kelas yang dipanggil saat objek dari kelas tersebut dihapus atau "dilepaskan" dari memori. Nama destruktor pada Python adalah __del__(), dan metode ini dapat digunakan untuk membersihkan sumber daya yang digunakan oleh objek, seperti menutup file atau koneksi database. Destruktor pada Python digunakan untuk membersihkan sumber daya yang digunakan oleh objek sebelum objek dihapus dari memori dan menjalankan kode atau operasi tertentu saat objek dihapus dari memori.

Contoh implementasi destruktor pada python :

```
1. class Mahasiswa:
2.     def __init__(self, nama, jurusan):
3.         self.nama = nama
4.         self.jurusan = jurusan
5.
6.     def __del__(self):
7.         print("Objek Mahasiswa dengan nama", self.nama, "telah dihapus")
```

```
8.
9. data = Mahasiswa("Bagas Satrio", "Informatika")
10. # objek mahasiswa1 akan dihapus dari memori dan destruktor akan
    dipanggil
11. del data
```

Pada contoh di atas, destruktor __del__() digunakan untuk mencetak pesan "Objek Mahasiswa dengan nama <nama> telah dihapus" saat objek dari kelas Mahasiswa dihapus dari memori dengan menggunakan fungsi del. Saat fungsi del data dijalankan, objek data akan dihapus dari memori dan destruktor __del__() akan dipanggil untuk menjalankan kode atau operasi yang didefinisikan di dalamnya.

F. SETTER DAN GETTER

Setter dan getter pada Python adalah metode khusus pada sebuah kelas yang digunakan untuk mengatur (set) dan mendapatkan (get) nilai dari atribut pada objek. Dalam Python, setter dan getter biasanya didefinisikan dengan menggunakan decorator @property dan @<nama atribut>.setter. Setter pada Python digunakan untuk mengatur nilai dari suatu atribut pada objek, melakukan validasi atau operasi tertentu saat nilai atribut diatur. Getter pada Python digunakan untuk mendapatkan nilai dari suatu atribut pada objek, menjalankan kode atau operasi tertentu saat nilai atribut didapatkan.

Contoh Implementasi Setter dan Getter pada python :

```
1. class Mahasiswa:
2.     def __init__(self, nama, jurusan):
3.         self.nama = nama
4.         self.jurusan = jurusan
5.
6.     @property
7.     def nama(self):
8.         return self._nama
9.
10.    @nama.setter
11.    def nama(self, value):
12.        if len(value) < 2:
13.            raise ValueError("Panjang nama minimal 2 karakter")
14.        self._nama = value
15.
16.    @property
17.    def jurusan(self):
18.        return self._jurusan
19.
20.    @jurusan.setter
21.    def jurusan(self, value):
22.        if value not in ["Informatika", "Mesin", "Pertambangan"]:
```

```
            raise ValueError("Jurusan tidak valid")
        self._jurusan = value

data = Mahasiswa("Bagas Satrio", "Informatika")
# mendapatkan nilai atribut jurusan pada objek data
print(data.jurusan)
```

Pada contoh di atas, setter dan getter pada Python digunakan untuk mengatur dan mendapatkan nilai dari atribut nama dan jurusan pada objek dari kelas Mahasiswa. Setter nama.setter digunakan untuk melakukan validasi bahwa nilai nama yang diatur minimal memiliki panjang 2 karakter. Setter jurusan.setter digunakan untuk melakukan validasi bahwa nilai jurusan yang diatur harus merupakan salah satu dari tiga nilai yang valid. Getter nama dan jurusan digunakan untuk mendapatkan nilai dari atribut nama dan jurusan pada objek. Penggunaan decorator @property pada setter dan getter menandakan bahwa metode tersebut adalah metode untuk mengatur dan mendapatkan nilai atribut.

G. DECORATOR

Decorator pada Python adalah fitur bahasa pemrograman yang memungkinkan kita untuk mengubah atau memperluas fungsi atau kelas dengan menambahkan fungsi lain tanpa harus mengubah kode asli dari fungsi atau kelas tersebut. Decorator pada dasarnya adalah fungsi yang mengambil fungsi atau kelas lain sebagai argumen dan mengembalikan fungsi atau kelas yang diperluas.

Contoh Implementasi decorator pada python :

```
class Mahasiswa:
    def __init__(self, nama, jurusan, umur = 20):
        self.nama = nama
        self.jurusan = jurusan
        self.umur = umur

    @property
    def umur(self):
        print ("Fungsi Umur Getter Dipanggil")
        return self.__umur

    @umur.setter
    def umur(self,x):
        print ("Fungsi Umur Setter Dipanggil")
        self.__umur = x


data = Mahasiswa("Bagas Satrio", "Informatika")

#Decorator
```

```
21.print(data.umur)
22.data.umur += 5
23.print(data.umur)
```

Method getter umur menggunakan decorator @property dan diimplementasikan pada method def umur(self). Method setter umur menggunakan decorator @umur.setter dan diimplementasikan pada method def umur(self,x). Saat method umur dipanggil dengan nilai x, maka akan menampilkan pesan "Fungsi Umur Setter Dipanggil" dan mengubah nilai dari atribut __umur menjadi nilai dari x. Jika dijalankan akan mendapatkan output seperti berikut:

```
Fungsi Umur Setter Dipanggil
Fungsi Umur Getter Dipanggil
20
Fungsi Umur Getter Dipanggil
Fungsi Umur Setter Dipanggil
Fungsi Umur Getter Dipanggil
25
```

BAB II

ABSTRAKSI DAN ENKAPSULASI (VISIBILITAS FUNGSI DAN VARIABEL, RELASI ANTAR KELAS)

A. ABSTRAKSI

Abstraksi adalah konsep yang penting dalam pengembangan perangkat lunak. Konsep ini memungkinkan kita untuk menyembunyikan detail yang tidak relevan dan fokus pada informasi penting yang relevan. Dalam pemrograman, abstraksi dapat diterapkan melalui konsep-konsep seperti kelas, fungsi, dan modul. Dalam Python, abstraksi dapat diterapkan melalui kelas. Abstraksi adalah konsep OOP dimana model yang dibuat hanya memperlihatkan atribut yang esensial dan menyembunyikan detail-detail yang tidak penting dari user yang berfungsi untuk mengurangi kompleksitas. Kelas adalah struktur dasar dalam pemrograman berorientasi objek yang digunakan untuk mewakili objek dalam program. Dalam kelas, kita dapat menentukan atribut dan metode yang akan dimiliki oleh objek yang akan dibuat. Contohnya, jika kita ingin membuat kelas untuk merepresentasikan sebuah biodata mahasiswa, kita dapat menentukan atribut seperti nama, nim, mata kuliah dan jumlah sks yang diambil, serta metode seperti menampilkan data Berikut adalah contoh implementasi abstraksi dalam kode Python:

```
1. class Mahasiswa:
2.
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         self.nama = nama
5.         self.nim = nim
6.         self.pbo = pbo
7.         self.jumsks = jumsks
8.     def tampil_data(self):
9.         print ("Nama = ", self.nama)
10.        print ("NIM = ", self.nim)
11.        print ("Kelas PBO = ", self.pbo)
12.        print ("Jumlah SKS = ", self.jumsks)
13.
14.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
15.data.tampil_data()
```

Pada contoh di atas, kita membuat kelas **“Mahasiswa”** dengan atribut **“nama”, “nim”, “pbo”, “jumsks”** serta metode **“tampil_data”**. Kemudian, kita membuat objek **“data”** yang merupakan instance dari kelas **“Mahasiswa”**. Dalam pemrograman, abstraksi sangat penting untuk membuat kode yang mudah dipahami, mudah diubah, dan mudah dikembangkan. Dengan abstraksi, kita dapat menghindari kesalahan dan mengurangi kompleksitas kode yang dapat mempengaruhi performa dan kualitas perangkat lunak.

B. ENKAPSULASI

Enkapsulasi (Encapsulation) adalah sebuah teknik dalam OOP yang mengizinkan kita untuk menyembunyikan detail dari sebuah atribut dalam sebuah class. 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. Untuk membatasi hak akses terhadap property dan method dalam suatu kelas, terdapat 3 jenis access modifier yang terdapat dalam python, yaitu public access, protected access, dan private access.

1. 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 atau dapat didefinisikan dengan secara langsung menuliskan nama dari atribut atau fungsi. Dalam sebuah objek, atribut atau fungsi yang bersifat public access dapat diakses di luar scope sebuah class.

   Contoh :

```
1. class Mahasiswa:
2.     #Constructor
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         #Public Variable
5.         self.nama = nama
6.         self.nim = nim
7.         self.pbo = pbo
8.         self.jumsks = jumsks
```

   Pada contoh kode program di atas, kita membuat sebuah kelas dengan nama **“Mahasiswa”**. Kelas tersebut memiliki 4 buah atribut yaitu **“nama”, “nim”, “pbo”** dan **“jumsks”** yang mana semuanya memiliki hak akses publik. Untuk membuktikannya kita coba akses ke-4 atribut di atas dari luar kelas:

```
1. class Mahasiswa:
2.     #Constructor
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         #Public Variable
5.         self.nama = nama
6.         self.nim = nim
7.         self.pbo = pbo
8.         self.jumsks = jumsks
9.
10.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
11.
12.#Akses Variable nama, nim, pbo dan jumsks dari luar kelas
```

```
13.print(f'Nama : {data.nama}')
14.print(f'Nim : {data.nim}')
15.print(f'Kelas PBO : {data.pbo}')
16.print(f'Jumlah SKS : {data.jumsks}')
```

Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama : Bagas Satrio
Nim : 121140077
Kelas PBO : RB
Jumlah SKS : 22
```

2. Protected Access Modifier
   Variabel atau atribut yang memiliki hak akses protected yang hanya bisa diakses secara terbatas oleh dirinya sendiri (yaitu di dalam internal kelas), dan juga dari kelas turunannya. Untuk mendefinisikan atribut dengan hak akses protected, kita harus menggunakan prefix underscore _ sebelum nama variabel. Contoh :

```
1. class Mahasiswa:
2.     #Constructor
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         #Protected Variable
5.         self._nama = nama
6.         self._nim = nim
7.         self._pbo = pbo
8.         self._jumsks = jumsks
9.
10.        #Protected Method
11.    def tampil_data(self):
12.        print ("Nama = ", self._nama)
13.        print ("NIM = ", self._nim)
14.        print ("Kelas PBO = ", self._pbo)
15.        print ("Jumlah SKS = ", self._jumsks)
16.
17.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
18.data.tampil_data()
```

Pada contoh kode program di atas, kita membuat sebuah kelas bernama **"Mahasiswa"**. Kemudian di dalam kelas tersebut, kita mendefinisikan 4 buah atribut bernama yaitu **"_nama", "_nim", "_pbo"** dan **"_jumsks"** yang mana hak akses dari atribut tersebut adalah protected karena nama variabelnya diawali oleh underscore (_). Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama : Bagas Satrio
Nim : 121140077
Kelas PBO : RB
Jumlah SKS : 22
```

3. Private Access Modifier
   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. Private Acces Modifier dapat didefinisikan dengan menambahkan double underscore (__).
   Contoh :

```
1. class Mahasiswa:
2.     #Constructor
3.     def __init__(self,nama,nim,pbo,jumsks):
4.         #Private Variable
5.         self.__nama = nama
6.         self.__nim = nim
7.         self.__pbo = pbo
8.         self.__jumsks = jumsks
9.
10.        #Private Method
11.    def tampil_data(self):
12.        print ("Nama = ", self.__nama)
13.        print ("NIM = ", self.__nim)
14.        print ("Kelas PBO = ", self.__pbo)
15.        print ("Jumlah SKS = ", self.__jumsks)
16.
17.data = Mahasiswa("Bagas Satrio",121140077,"RB",22)
18.data.tampil_data()
```

Pada contoh kode program di atas, kita membuat sebuah kelas bernama **“Mahasiswa”**. Kemudian di dalam kelas tersebut, kita mendefinisikan 4 buah atribut bernama yaitu **“_nama”, “_nim”, “_pbo”** dan **“__jumsks”.** Dan karena nama tersebut diawali dua buah underscore, maka ia tidak bisa diakses kecuali dari dalam kelas **“Mahasiswa”** saja. Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama : Bagas Satrio
Nim : 121140077
Kelas PBO : RB
Jumlah SKS : 22
```

BAB III

PEWARISAN DAN POLIMORFISME

(OVERLOADING, OVERRIDING, DYNAMIC CAST)

A. INHERITANCE (PEWARISAN)

Inheritance dalam pemrograman komputer adalah sebuah konsep di mana sebuah objek (kelas) dapat memperoleh sifat atau perilaku dari objek lain yang lebih generik (kelas induk atau superclass). Dalam konsep ini, sebuah kelas dapat mewarisi sifat dan perilaku dari kelas induk, dan juga dapat menambahkan sifat atau perilaku yang unik untuk kelas itu sendiri. Dengan menggunakan konsep inheritance, kita dapat membuat hierarki kelas yang terorganisir dengan baik, dengan kelas yang lebih umum sebagai kelas induk, dan kelas yang lebih spesifik sebagai kelas turunan atau subclass. Hal ini dapat memudahkan pengembangan program, karena kita dapat membagi kode program ke dalam bagian-bagian yang lebih kecil dan mudah diatur. Selain itu, inheritance juga dapat meminimalkan duplikasi kode dan meningkatkan efisiensi penggunaan memori.
Berikut merupakan sintaks dasar inheritance:

```
class Parent:
    pass

class Child(Parent):
    pass
```

Contoh :

```
#Class Parent
class Mahasiswa:
    #Constructor
    def __init__(self,nama,nim,pbo,jumsks):
        self.nama = nama
        self.nim = nim
        self.pbo = pbo
        self.jumsks = jumsks
    #Atribut Parent
    def tampil_data(self):
        print ("Nama = ", self.nama)
        print ("NIM = ", self.nim)
        print ("Kelas PBO = ", self.pbo)
        print ("Jumlah SKS = ", self.jumsks)

#Class Turunan atau Child dari Class Mahasiswa
class Orang(Mahasiswa):
    pass
```

```
19.
20.data = Orang("Bagas Satrio",121140077,"RB",22)
21.data.tampil_data()
```

Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama : Bagas Satrio
Nim : 121140077
Kelas PBO : RB
Jumlah SKS : 22
```

B. INHERITANCE IDENTIK

Inheritance identik merupakan pewarisan yang menambahkan constructor pada class child sehingga class child memiliki constructornya sendiri tanpa menghilangkan constructor pada class parentnya. Metode ini ditandai dengan adanya class child yang menggunakan constructor dan menggunakan kata kunci super().
Contoh :

```
1.  #Class Parent
2.  class Mahasiswa:
3.      #Constructor
4.      def __init__(self,nama,nim,pbo,jumsks):
5.          self.nama = nama
6.          self.nim = nim
7.          self.pbo = pbo
8.          self.jumsks = jumsks
9.      #Atribut Parent
10.     def tampil_data(self):
11.         print ("Nama = ", self.nama)
12.         print ("NIM = ", self.nim)
13.         print ("Kelas PBO = ", self.pbo)
14.         print ("Jumlah SKS = ", self.jumsks)
15.
16.#Class Turunan atau Child dari Class Mahasiswa
17.class Orang(Mahasiswa):
18.     def __init__(self,nama,nim,pbo,jumsks,universitas):
19.         #Inheritance Identik
20.         super().__init__(nama,nim,pbo,jumsks)
21.         self.universitas = universitas
22.     #Atribut Child
23.     def tampil_data(self):
24.         print ("Nama = ", self.nama)
25.         print ("NIM = ", self.nim)
26.         print ("Kelas PBO = ", self.pbo)
```

```
27.         print ("Jumlah SKS = ", self.jumsks)
28.         print ("Universitas = ",self.universitas)
29.
30.data = Orang("Bagas Satrio",121140077,"RB",22,"Institut Teknologi
   Sumatera")
31.data.tampil_data()
```

Pada child class dapat ditambahkan beberapa fitur tambahan baik atribut maupun method sehingga child class tidak identik dengan parent class. Dari contoh program di atas terlihat bahwa terdapat penambahan karakteristik pada child class yaitu berupa atribut"**universitas"**. Jika dijalankan akan mendapatkan output seperti berikut:

```
Nama =  Bagas Satrio
NIM =  121140077
Kelas PBO =  RB
Jumlah SKS =  22
Universitas =  Institut Teknologi Sumatera
```

C. POLYMORPHISM

Polymorphism dalam pemrograman Python adalah kemampuan objek untuk menunjukkan berbagai bentuk (tipe data) atau perilaku yang berbeda dalam konteks yang berbeda. Dalam bahasa Python, Polymorphism dapat diimplementasikan melalui konsep Overloading atau Overriding.

1. Overriding
   Overriding adalah konsep di mana subclass dapat mengganti implementasi metode yang ada di superclass. Dalam konsep ini, metode dengan nama yang sama akan diganti oleh metode yang didefinisikan di dalam subclass. Dalam bahasa Python, Overriding dapat dilakukan dengan cara menulis ulang metode superclass pada subclass dengan implementasi yang berbeda.

   Contoh implementasi Overloading dalam Python :

```
1. class AlatMusik:
2.     def tampil(self):
3.         print("Ini Alat Musik")
4.
5. class Gitar(AlatMusik):
6.     def tampil(self):
7.         print("Ini Alat Musik Yang Bernama Gitar")
8.
9. data = Gitar()
10.data.tampil()
```

   Pada contoh terlihat bahwa terdapat parent class yaitu class **"AlatMusik"** dengan method **"tampil"**. Kemudian terdapat child class yaitu class **"Gitar"** yang memiliki

method yang sama dengan method pada parent class. Dengan begitu ketika melakukan instansiasi objek child class dan memanggil method tampil() maka yang akan dipanggil ialah method pada child class, method pada parent class tidak berlaku. Jika dijalankan akan mendapatkan output seperti berikut:

```
Ini Alat Musik Yang Bernama Gitar
```

2. Overloading

Overloading adalah konsep di mana beberapa fungsi dapat memiliki nama yang sama tetapi memiliki parameter dan implementasi yang berbeda. Python mendukung konsep Overloading pada fungsi melalui penggunaan *args atau **kwargs sebagai parameter untuk fungsi yang dapat menerima jumlah parameter yang berbeda.
Contoh implementasi Overloading dalam Python :

```
1. class Kalkulator:
2.     def penjumlahan(self, a, b, *args):
3.         if args:
4.             return a + b + sum(args)
5.         else:
6.             return a + b
7.
8. data = Kalkulator()
9. print("Hasil Penjumlahan = ", data.penjumlahan(1, 2))
10.print("Hasil Penjumlahan = ", data.penjumlahan(1, 2, 3))
11.print("Hasil Penjumlahan = ", data.penjumlahan(1, 2, 3, 4, 5))
```

Pada contoh di atas, terdapat sebuah kelas **"Kalkulator"** yang memiliki sebuah metode penjumlahan dengan parameter a, b, dan *args yang dapat menerima jumlah parameter yang berbeda. Jika hanya dua parameter yang diberikan, metode akan mengembalikan hasil penjumlahan dari dua parameter tersebut. Namun, jika terdapat lebih dari dua parameter, metode akan menghitung jumlah dari semua parameter tersebut dan mengembalikan hasilnya. Jika dijalankan akan mendapatkan output seperti berikut:

```
Hasil Penjumlahan =  3
Hasil Penjumlahan =  6
Hasil Penjumlahan =  15
```

D. MULTIPLE INHERITANCE

Multiple Inheritance adalah konsep di mana sebuah kelas dapat mewarisi sifat atau karakteristik dari dua atau lebih kelas induk (parent class). Dalam konsep ini, kelas anak (subclass) dapat memiliki semua metode dan atribut yang dimiliki oleh kelas induknya. Dalam Multiple Inheritance, kelas anak dapat mewarisi atribut dan metode dari dua kelas induk atau lebih. Jika terdapat dua atau lebih kelas induk yang memiliki metode atau atribut

dengan nama yang sama, maka kelas anak harus menentukan cara menyelesaikan konflik tersebut, misalnya dengan melakukan override terhadap metode atau atribut tersebut. Bentuk syntax multiple inheritance adalah sebagai berikut :

```
class Parent1:
    pass

class Parent2:
    pass

class MultipleInheritance(Parent1, Parent2):
    pass
```

Contoh implementasi Multiple Inheritance pada Python :

```
class Mahasiswa:
    def __init__(self, nama, alamat, tempat_lahir):
        self.nama = nama
        self.alamat = alamat
        self.tempat_lahir = tempat_lahir

    def data_mahasiswa(self):
        print("Biodata Mahasiswa")
        print("Nama : ", self.nama)
        print("Alamat : ", self.alamat)
        print("Tempat Lahir : ", self.tempat_lahir)

class Akademik:
    def __init__(self, nim, jurusan, fakultas):
        self.nim = nim
        self.jurusan = jurusan
        self.fakultas = fakultas

    def info_akademik(self):
        print("Biodata Akademik")
        print("NIM : ", self.nim)
        print("Jurusan : ", self.jurusan)
        print("Fakultas : ", self.fakultas)

class DataMahasiswa(Mahasiswa, Akademik):
    def __init__(self, nama, alamat, tempat_lahir, nim, jurusan,
fakultas):
        Mahasiswa.__init__(self, nama, alamat, tempat_lahir)
        Akademik.__init__(self, nim, jurusan, fakultas)

```

```
30.     def info_mahasiswa(self):
31.         print("Biodata Data Mahasiswa")
32.         print()
33.         super().data_mahasiswa()
34.         print()
35.         super().info_akademik()
36.
37. # Membuat objek dari kelas DataMahasiswa
38. mhs = DataMahasiswa("Bagas Satrio", "Kemiling, Bandar Lampung",
    "Pringsewu", "121140077", "Teknik Informatika", "Fakultas Teknik")
39.
40. # Memanggil metode info_mahasiswa dari kelas DataMahasiswa
41. mhs.info_mahasiswa()
```

Pada contoh di atas, terdapat kelas **"Mahasiswa"** dan **"Akademik"** sebagai kelas induk atau parent class, dan kelas **"DataMahasiswa"** sebagai kelas anak atau subclass yang mewarisi sifat atau karakteristik dari kelas **"Mahasiswa"** dan **"Akademik"**. Kelas **"DataMahasiswa"** merupakan contoh dari Multiple Inheritance, karena mewarisi sifat dari dua kelas induk, yaitu **"Mahasiswa"** dan **"Akademik"**. Dalam kelas **"DataMahasiswa"**, terdapat metode **"info_mahasiswa"** yang melakukan override terhadap metode yang sama pada kelas induknya. Dalam metode **"info_mahasiswa"** pada kelas **"DataMahasiswa"**, terdapat pemanggilan metode **"data_mahasiswa"** pada kelas **"Mahasiswa"** dan metode **"info_akademik"** pada kelas Akademik. Jika dijalankan akan mendapatkan output seperti berikut :

```
Biodata Data Mahasiswa

Biodata Mahasiswa
Nama :  Bagas Satrio
Alamat :  Kemiling, Bandar Lampung
Tempat Lahir :  Pringsewu

Biodata Akademik
NIM :  121140077
Jurusan :  Teknik Informatika
Fakultas :  Fakultas Teknik
```

E. METHOD RESOLUTION ORDER

Method Resolution Order (MRO) adalah urutan pencarian metode yang dimiliki oleh sebuah objek pada saat terjadi multiple inheritance di Python. MRO menentukan urutan pewarisan metode dari class induk ke class anak saat terdapat beberapa class yang diwarisi oleh class anak. Pada Python, MRO diatur menggunakan algoritma C3. Algoritma ini memastikan bahwa urutan pencarian metode pada multiple inheritance di Python memiliki keteraturan dan tidak menghasilkan ambiguitas. Algoritma C3 bekerja dengan

menggabungkan daftar urutan pewarisan metode dari setiap class dalam hierarki class yang diwarisi oleh class anak. Kemudian, algoritma C3 menyelesaikan daftar yang telah digabungkan tersebut dengan mengikuti tiga aturan dasar:

1. Setiap class harus muncul hanya satu kali dalam daftar MRO. Ini termasuk class anak dan class induk, serta class yang diwarisi oleh class induk.
2. Urutan pewarisan metode harus mempertahankan urutan linier dari class induk.
3. Jika terdapat dua class dengan urutan pewarisan metode yang sama, maka urutan class yang diberikan dalam definisi multiple inheritance harus diikuti.

Demonstrasi MRO :

```
1. class X:
2.     pass
3.
4. class Y:
5.     pass
6.
7. class Z:
8.     pass
9.
10.class A(X,Y):
11.    pass
12.
13.class B(Y,Z):
14.    pass
15.
16.class M(B, A, Z):
17.    pass
```

```
# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
#  <class '__main__.A'>, <class '__main__.X'>,
#  <class '__main__.Y'>, <class '__main__.Z'>,
```

Contoh implementasi Method Resolution Order pada Python :

```
1. class X:
2.     def method(self):
3.         print("Metode X Dipanggil")
4.
5. class Y:
6.     def method(self):
7.         print("Metode Y Dipanggil")
8.
9. class A(X,Y):
```

```
    pass

class B(Y,X):
    pass

c=A()
c.method()

d=B()
d.method()
```

Pada contoh di atas, class **"A"** mewarisi sifat dari class **"X"** dan **"Y"**. Karena kedua class induk tersebut memiliki metode dengan nama yang sama (method), maka MRO akan menentukan urutan pemanggilan metode tersebut. Berdasarkan aturan MRO, metode dari class **"X"** akan dipanggil terlebih dahulu sebelum metode dari class **"Y"**, sehingga output yang dihasilkan adalah **"Metode X Dipanggil"**. Begitu juga dengan class **"B"** yang mewarisi sifat dari class **"Y"** dan **"X"**. Maka berdasarkan aturan MRO, metode dari class **"Y"** akan dipanggil terlebih dahulu sebelum metode dari class **"Y"** sehingga output yang akan dihasilkan adalah **"Metode Y Dipanggil"**. Jika dijalankan akan mendapatkan output seperti berikut :

```
Metode X Dipanggil
Metode Y Dipanggil
```

F. DYNAMIC CAST

Dynamic cast atau type conversion adalah proses mengubah nilai dari satu tipe data ke tipe data lainnya seperti dari string ke int atau sebaliknya. Ada 2 tipe konversi yaitu:

1. Implisit
   proses konversi tipe data secara otomatis oleh interpreter Python. Ketika kita melakukan operasi matematika atau operasi lain yang melibatkan tipe data yang berbeda, Python akan secara otomatis mengubah salah satu tipe data agar sesuai dengan yang lain. Contoh casting implisit adalah ketika kita menambahkan integer dan float, maka Python akan mengubah integer menjadi float agar dapat dilakukan operasi penjumlahan. Contoh Implementasi Implisit Pada Pyhton :

```
num_int = 111
num_float = 1.111

num = num_int + num_float

print("Tipe Data num_int : ",type(num_int))
print("Tipe Data num_float : ",type(num_float))
```

```
9. print("Jumlah Penjumlahan = ",num)
10.print("Type Data Penjumlahan(num) : ",type(num))
```

Pada contoh di atas, variabel **"num_int"** memiliki tipe data integer, sedangkan variabel **"num_float"** memiliki tipe data float. Ketika variabel **"num_int"** dan **"num_float"** dijumlahkan, Python akan secara otomatis mengkonversi tipe data **"num_int"** menjadi float dan melakukan operasi penjumlahan, sehingga variabel **"num"** akan memiliki tipe data float. Proses konversi tipe data ini terjadi secara implisit, tanpa perlu melakukan operasi casting. Jika dijalankan akan mendapatkan output seperti berikut :

```
Tipe Data num_int :  <class 'int'>
Tipe Data num_float :  <class 'float'>
Jumlah Penjumlahan =  112.111
Type Data Penjumlahan(num) :  <class 'float'>
```

2. Eksplisit

Casting eksplisit adalah proses konversi tipe data secara manual oleh programmer dengan menggunakan konstruktor tipe data. Dalam Python, terdapat beberapa konstruktor tipe data yang digunakan untuk casting eksplisit, antara lain:

- int() untuk mengkonversi menjadi tipe data integer
- float() untuk mengkonversi menjadi tipe data float
- str() untuk mengkonversi menjadi tipe data string

Contoh Implementasi Eksplisit Pada Pyhton :

```
1. num_int = 111
2. num_str = "111"
3.
4. print("Tipe Data num_int : ",type(num_int))
5. print("Tipe Data num_str Sebelum Type Casting : ",type(num_str))
6.
7. num_str = int(num_str)
8. print("Tipe Data num_str Setelah Type Casting : ",type(num_str))
9.
10.num_sum = num_int + num_str
11.
12.print("Jumlah Penjumlahan : ",num_sum)
13.print("Tipe Data Penjumlahan(num_sum) : ",type(num_sum))
```

Jika dijalankan akan mendapatkan output seperti berikut :

```
Tipe Data num_int :  <class 'int'>
Tipe Data num_str Sebelum Type Casting :  <class 'str'>
Tipe Data num_str Setelah Type Casting :  <class 'int'>
Jumlah Penjumlahan :  222
Tipe Data Penjumlahan(num_sum) :  <class 'int'>
```

G. CASTING

1. Downcasting

   Downcasting adalah proses konversi objek dari kelas induknya (superclass) ke turunan kelasnya. Downcasting harus dilakukan secara eksplisit dengan menggunakan operator khusus yaitu **as**. Namun, downcasting hanya dapat dilakukan jika objek tersebut sebenarnya adalah objek dari turunan kelas.

   Contoh implementasi downcasting pada Python:

```
1. class Mahasiswa:
2.     def __init__(self, nama, alamat):
3.         self.nama = nama
4.         self.alamat = alamat
5.
6.     def data_mahasiswa(self):
7.         print("Biodata Mahasiswa")
8.         print("Nama : ", self.nama)
9.         print("Alamat : ", self.alamat)
10.        #Downcasting
11.        print("Jurusan : ",(self.jurusan))
12.
13.class Data(Mahasiswa):
14.    def __init__(self, nama, alamat, jurusan):
15.        super().__init__(nama, alamat)
16.        self.jurusan = jurusan
17.
18.bio = Data("Bagas Satrio", "Kemiling, Bandar Lampung", "Teknik
   Informatika")
19.bio.data_mahasiswa()
```

   Jika dijalankan akan mendapatkan output sebagai berikut :

```
Biodata Mahasiswa
Nama :  Bagas Satrio
Alamat :  Kemiling, Bandar Lampung
Jurusan :  Teknik Informatika
```

2. Upcasting

Upcasting adalah proses konversi objek dari turunan kelas ke kelas induknya (superclass). Upcasting dilakukan secara otomatis oleh Python, tanpa perlu menggunakan operator khusus. Upcasting terjadi ketika objek dari turunan kelas di-assign ke variabel yang bertipe kelas induknya.

Contoh Impelentasi upcasting pada Python :

```
1. class Mahasiswa:
2.     #Upcasting
3.     alamat = "Pringsewu"
4.     def __init__(self, nama, alamat):
5.         self.nama = nama
6.         self.alamat = alamat
7.
8.     def data_mahasiswa(self):
9.         print("Biodata Mahasiswa")
10.        print("Nama : ", self.nama)
11.        print("Alamat : ", self.alamat)
12.        print("Jurusan : ",(self.jurusan))
13.
14.class Data(Mahasiswa):
15.    def __init__(self, nama, alamat, jurusan):
16.        #Upcasting
17.        super().__init__(nama, alamat)
18.        self.jurusan = jurusan
19.
20.    def data_mahasiswa(self):
21.        print("Biodata Mahasiswa")
22.        print("Nama : ", self.nama)
23.        #Upcasting
24.        print("Alamat : ", super().alamat)
25.        print("Jurusan : ",(self.jurusan))
26.bio = Data("Bagas Satrio", "Kemiling, Bandar Lampung", "Teknik
   Informatika")
27.bio.data_mahasiswa()
```

Jika dijalankan akan mendapatkan output seperti berikut :

```
Biodata Mahasiswa
Nama :  Bagas Satrio
Alamat :  Pringsewu
Jurusan :  Teknik Informatika
```

3. Type Casting

   Konversi tipe kelas agar memiliki sifat/perilaku tertentu yang secara default tidak dimiliki kelas tersebut. Karena Python merupakan bahasa pemrograman berorientasi objek, maka semua variabel atau instansi di Python pada dasarnya merupakan objek(kelas) yang sifat/perilakunya dapat dimanipulasi jika dibutuhkan (umumnya melalui magic method).

   Contoh implementasi Type Casting pada python :

```
1. class Mahasiswa:
2.     def __init__(self, nama, nim, matkul):
3.         self.__nama = nama
4.         self.__nim = nim
5.         self.__matkul = matkul
6.
7.     def __str__(self):
8.         return f"{self.__nama} ({self.__nim}) merupakan mahasiswa
   kelas {self.__matkul}"
9.     def __int__(self):
10.        return self.__nim
11.
12.bio = Mahasiswa("Bagas Satrio", 121140077, "PBO RB")
13.print(bio)
14.print(int(bio) == 121140077)
```

   Jika dijalankan akan mendapatkan output seperti berikut :

```
Bagas Satrio (121140077) merupakan mahasiswa kelas PBO RB
True
```