# PRAKTIKUM MODUL 6
# PEMOGRAMAN BERBASIS OBJEK
# KELAS ABSTRAK, INTERFACE, DAN METACLASS

Oleh:
Fitra Ilyasa 120140048


**TEKNIK INFORMATIKA**
**INSTITUT TEKNOLOGI SUMATERA**
**LAMPUNG SELATAN**
**2022**

# BAB I

# ABSTRAK

Metode abstrak adalah metode yang dideklarasikan oleh antarmuka Python, tetapi mungkin tidak memiliki implementasi yang berguna. Metode abstrak harus ditimpa oleh kelas konkret yang mengimplementasikan antarmuka yang dimaksud. Untuk membuat metode abstrak dengan Python, Anda menambahkan dekorator @abc.abstractmethod ke metode antarmuka. Dalam contoh berikutnya, Anda memperbarui FormalParserInterface untuk menyertakan metode abstrak .load_data_source() dan .extract_text():

```
class FormalParserInterface(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(cls, subclass):
        return (hasattr(subclass, 'load_data_source') and
                callable(subclass.load_data_source) and
                hasattr(subclass, 'extract_text') and
                callable(subclass.extract_text) or
                NotImplemented)

    @abc.abstractmethod
    def load_data_source(self, path: str, file_name: str):
        """Load in the data set"""
        raise NotImplementedError

    @abc.abstractmethod
    def extract_text(self, full_file_path: str):
        """Extract text from the data set"""
        raise NotImplementedError

class PdfParserNew(FormalParserInterface):
    """Extract text from a PDF."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides FormalParserInterface.load_data_source()"""
        pass
```

```
class EmlParserNew(FormalParserInterface):
    """Extract text from an email."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides FormalParserInterface.load_data_source()"""
        pass

    def extract_text_from_email(self, full_file_path: str) -> dict:
        """A method defined only in EmlParser.
        Does not override FormalParserInterface.extract_text()
        """
        pass
```

Dalam contoh di atas, Anda akhirnya membuat antarmuka formal yang akan menimbulkan kesalahan saat metode abstrak tidak diganti. Instance PdfParserNew, pdf_parser, tidak akan memunculkan kesalahan apa pun, karena PdfParserNew dengan benar menimpa metode abstrak FormalParserInterface. Namun, EmlParserNew akan memunculkan kesalahan:

```
>>> pdf_parser = PdfParserNew()
>>> eml_parser = EmlParserNew()
Traceback (most recent call last):
  File "real_python_interfaces.py", line 53, in <module>
    eml_interface = EmlParserNew()
TypeError: Can't instantiate abstract class EmlParserNew with abstract methods
```

Seperti yang Anda lihat, pesan traceback memberi tahu Anda bahwa Anda belum mengganti semua metode abstrak. Ini adalah perilaku yang Anda harapkan saat membangun antarmuka Python formal.

# BAB II
# INTERFACE

Interface merupakan metode yang digunakan supaya memungkinkan program untuk berbagi konstanta di dalam sejumlah kelas atau dapat juga berfungsi untuk mengartikan kontrak. Pada tingkat tinggi, antarmuka bertindak sebagai cetak biru untuk merancang kelas. Seperti kelas, antarmuka mendefinisikan metode. Tidak seperti kelas, metode ini abstrak. Metode abstrak adalah metode yang hanya didefinisikan oleh antarmuka. Itu tidak menerapkan metode. Ini dilakukan oleh kelas, yang kemudian mengimplementasikan antarmuka dan memberikan makna konkret pada metode abstrak antarmuka.

## Informal Interface

Dalam keadaan tertentu, Anda mungkin tidak memerlukan aturan ketat dari antarmuka Python formal. Sifat dinamis Python memungkinkan Anda untuk mengimplementasikan antarmuka informal. Antarmuka Python informal adalah kelas yang mendefinisikan metode yang dapat diganti, tetapi tidak ada penegakan yang ketat.

Dalam contoh berikut, Anda akan mengambil perspektif seorang insinyur data yang perlu mengekstrak teks dari berbagai jenis file tidak terstruktur yang berbeda, seperti PDF dan email. Anda akan membuat antarmuka informal yang mendefinisikan metode yang akan ada di kelas beton PdfParser dan EmlParser:

```
class InformalParserInterface:
    def load_data_source(self, path: str, file_name: str) -> str:
        """Load in the file for extracting text."""
        pass

    def extract_text(self, full_file_name: str) -> dict:
        """Extract text from the currently loaded file."""
        pass
```

InformalParserInterface mendefinisikan dua metode .load_data_source() dan .extract_text(). Metode-metode ini didefinisikan tetapi tidak diimplementasikan. Implementasi akan terjadi setelah Anda membuat kelas konkret yang mewarisi dari InformalParserInterface.

## Formal Interface

Antarmuka informal dapat berguna untuk proyek dengan basis kode kecil dan jumlah pemrogram terbatas. Namun, antarmuka informal akan menjadi pendekatan yang salah untuk

aplikasi yang lebih besar. Untuk membuat antarmuka Python formal, Anda memerlukan beberapa alat lagi dari modul abc Python.

Menggunakan abc.ABCMeta Untuk menerapkan instantiasi subkelas dari metode abstrak, Anda akan menggunakan ABCMeta bawaan Python dari modul abc. Kembali ke antarmuka UpdatedInformalParserInterface Anda, Anda membuat metaclass Anda sendiri, ParserMeta, dengan metode dunder yang diganti .__instancecheck__() dan .__subclasscheck__(). Daripada membuat metaclass Anda sendiri, Anda akan menggunakan abc.ABCMeta sebagai metaclass. Kemudian, Anda akan menimpa .__subclasshook__() menggantikan .__instancecheck__() dan .__subclasscheck__(), karena ini menciptakan implementasi yang lebih andal dari metode dunder ini.

```
import abc

class FormalParserInterface(metaclass=abc.ABCMeta):
    @classmethod
    def __subclasshook__(cls, subclass):
        return (hasattr(subclass, 'load_data_source') and
                callable(subclass.load_data_source) and
                hasattr(subclass, 'extract_text') and
                callable(subclass.extract_text))

class PdfParserNew:
    """Extract text from a PDF."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides FormalParserInterface.load_data_source()"""
        pass

    def extract_text(self, full_file_path: str) -> dict:
        """Overrides FormalParserInterface.extract_text()"""
        pass
```

```
class EmlParserNew:
    """Extract text from an email."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides FormalParserInterface.load_data_source()"""
        pass

    def extract_text_from_email(self, full_file_path: str) -> dict:
        """A method defined only in EmlParser.
        Does not override FormalParserInterface.extract_text()
        """
        pass
```

Jika Anda menjalankan issubclass() pada PdfParserNew dan EmlParserNew, maka issubclass() akan mengembalikan True dan False, masing-masing.

# BAB III
# METACLASS

Menggunakan Metaclass Idealnya, Anda ingin issubclass(EmlParser, InformalParserInterface) mengembalikan False ketika kelas pelaksana tidak mendefinisikan semua metode abstrak antarmuka. Untuk melakukan ini, Anda akan membuat metaclass bernama ParserMeta. Anda akan mengganti dua metode dunder:

1. .__instancecheck__()
2. .__subclasscheck__()

Di blok kode di bawah ini, Anda membuat kelas bernama UpdatedInformalParserInterface yang dibangun dari metaclass ParserMeta:

```
class ParserMeta(type):
    """A Parser metaclass that will be used for parser class creation.
    """
    def __instancecheck__(cls, instance):
        return cls.__subclasscheck__(type(instance))

    def __subclasscheck__(cls, subclass):
        return (hasattr(subclass, 'load_data_source') and
                callable(subclass.load_data_source) and
                hasattr(subclass, 'extract_text') and
                callable(subclass.extract_text))

class UpdatedInformalParserInterface(metaclass=ParserMeta):
    """This interface is used for concrete classes to inherit from.
    There is no need to define the ParserMeta methods as any class
    as they are implicitly made available via .__subclasscheck__().
    """
    pass
```

Sekarang ParserMeta dan UpdatedInformalParserInterface telah dibuat, Anda dapat membuat implementasi konkret Anda. Pertama, buat kelas baru untuk mengurai PDF yang disebut PdfParserNew:

```
class PdfParserNew:
    """Extract text from a PDF."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides UpdatedInformalParserInterface.load_data_source()"""
        pass

    def extract_text(self, full_file_path: str) -> dict:
        """Overrides UpdatedInformalParserInterface.extract_text()"""
        pass
```

Di sini, PdfParserNew menimpa .load_data_source() dan .extract_text(), jadi issubclass(PdfParserNew, UpdatedInformalParserInterface) harus mengembalikan True.

Di blok kode berikutnya, Anda memiliki implementasi baru dari parser email yang disebut EmlParserNew:

```
class EmlParserNew:
    """Extract text from an email."""
    def load_data_source(self, path: str, file_name: str) -> str:
        """Overrides UpdatedInformalParserInterface.load_data_source()"""
        pass

    def extract_text_from_email(self, full_file_path: str) -> dict:
        """A method defined only in EmlParser.
        Does not override UpdatedInformalParserInterface.extract_text()
        """
        pass
```

Di sini, Anda memiliki metaclass yang digunakan untuk membuat UpdatedInformalParserInterface. Dengan menggunakan metaclass, Anda tidak perlu mendefinisikan subclass secara eksplisit. Sebagai gantinya, subclass harus mendefinisikan metode yang diperlukan. Jika tidak, maka issubclass(EmlParserNew, UpdatedInformalParserInterface) akan mengembalikan False.

Menjalankan issubclass() pada kelas konkret Anda akan menghasilkan yang berikut:

```
>>> issubclass(PdfParserNew, UpdatedInformalParserInterface)
True

>>> issubclass(EmlParserNew, UpdatedInformalParserInterface)
False
```

Seperti yang diharapkan, EmlParserNew bukan subkelas dari UpdatedInformalParserInterface karena .extract_text() tidak didefinisikan di EmlParserNew.

Sekarang, mari kita lihat MRO:

```
>>> PdfParserNew.__mro__
(<class '__main__.PdfParserNew'>, <class 'object'>)
```

Seperti yang Anda lihat, UpdatedInformalParserInterface adalah superclass dari PdfParserNew, tetapi tidak muncul di MRO. Perilaku yang tidak biasa ini disebabkan oleh fakta bahwa UpdatedInformalParserInterface adalah kelas dasar virtual PdfParserNew.

# BAB IV
# KESIMPULAN

Dapat disimpulkan bahwa interface merupakan metode yang digunakan supaya memungkinkan program untuk berbagi konstanta di dalam sejumlah kelas atau dapat juga berfungsi untuk mengartikan kontrak. Penggunaan interface ini dapat dilakukan oleh kelas, yang kemudian akan diimplementasikan untuk antarmuka dan memberikan makna konkret pada metode abstrak antarmuka.

Metode abstrak adalah metode yang dideklarasikan oleh antarmuka Python, tetapi mungkin tidak memiliki implementasi yang berguna. Metode abstrak harus ditimpa oleh kelas konkret yang mengimplementasikan antarmuka yang dimaksud. Untuk membuat metode abstrak dengan Python, Anda menambahkan dekorator @abc.abstractmethod ke metode antarmuka.

Sementara itu, kelas konkret merupakan kelas yang biasanya digunakan tanpa konsep abstrak, sehingga dapat diinstansiasikan secara langsung objek dari kelas tersebut. Penggunaan dari kelas konkret itu bisa dilakukan ketika menjalankan, menghentikan, menyalakan sebuah mobil yang dapat terlihat langsung oleh pengguna.

Terakhir metaclass merupakan kelas spesial, dimana mengembalikan False ketika kelas pelaksana tidak mendefinisikan semua metode abstrak antarmuka.

## DAFTAR PUSTAKA

https://realpython.com/python-interface/

https://auftechnique.com/4-pillar-pemrograman-berorientasi-objek/

https://koding.alza.web.id/object-oriented-programming-oop/

https://www.pintaar.com/course/c/34

https://jagongoding.com/python/menengah/oop/konsep/