01076105, 01075106

Object Oriented Programming

Object Oriented Programming Project

Polymorphism, Multiple Inheritance

Polymorphism

- คุณสมบัติข้อที่ 4 ของ OOP คือ polymorphism
- คำนี้ มาจากคำว่า “poly” แปลว่าหลาย กับ “morph” แปลว่าเปลี่ยน รวมแล้ว แปลว่า เปลี่ยนได้หลายแบบ
- คำนี้หมายถึง การที่ operation หรือ method ใดๆ สามารถใช้งานกับ object ที่หลากหลายได้ เช่น จากรูป จะเห็นได้ว่า operator * สามารถใช้ได้กับ ตัวเลข string list โดยมีพฤติกรรมที่แตกต่างกันไปในแต่ละประเภท

```
l = [2, "1", 3, 4, [1]]

for shape in l:
    print(shape*2)
```

Polymorphism

- อีกตัวอย่าง คือ การเล่นไฟล์เพลง audio_file.play()

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")
        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
        print("playing {} as wav".format(self.filename))

mp3 = MP3File("myfile.mp3")
wav = WavFile("myfile.wav")
mp3.play()
wav.play()
```

Polymorphism

- polymorphism จะเกิดขึ้นเมื่อ **Interface** ของคลาส **เหมือนกัน**
- คลาส AudioFile เป็น base class มี constructor แต่คลาส MP3File และ WavFile ที่ Inherit มา ไม่มี constructor จึงใช้ constructor ของ base class
- Constructor มีหน้าที่ตรวจสอบชนิดของไฟล์ และ กำหนดชื่อไฟล์
- จะเห็นว่า method play() ในแต่ละคลาสจะทำงานต่างกัน ในไฟล์แต่ละประเภท
- ดังนั้นไฟล์แต่ละประเภท จึงถูกจัดการด้วยวิธีการที่แตกต่างกัน เรียกว่า polymorphism
- ความสามารถ polymorphism จะเกิดขึ้นได้ ต้องมี Inheritance มาก่อน
- เราอาจกล่าวได้ว่า polymorphism คือการใช้ Interface ร่วมกันของข้อมูลที่แตกต่างกัน

Polymorphism : magic method

- เราอาจออกแบบให้ Class ของเรา ตอบสนองกับ เครื่องหมาย +-*/ หรือ in
- หรือตอบสนองกับ subscript หรือ slice หรือ loop (คล้ายกับ list) ได้
- ฟังก์ชัน __add__ อยู่ในกลุ่มที่เรียกว่า magic method หรือ dunder (double under)

```
>>> var1 = 'hello '
>>> var2 = 'world'
>>> var1 + var2
'hello world'
>>> var1.__add__(var2)
'hello world'
```

Polymorphism : magic method

- สร้างคลาสที่เมื่อกำหนดความยาวด้วยหน่วยหนึ่ง สามารถแสดงในหน่วยอื่นได้

```
class Length:
    __metric = {"mm" : 0.001, "cm" : 0.01, "m" : 1, "km" : 1000,
                "in" : 0.0254, "ft" : 0.3048, "yd" : 0.9144,
                "mi" : 1609.344 }
    def __init__(self, value, unit = "m" ):
        self.value = value
        self.unit = unit
    def Converse2Metres(self):
        return self.value * Length.__metric[self.unit]
    def __add__(self, other):
        l = self.Converse2Metres() + other.Converse2Metres()
        return Length(l / Length.__metric[self.unit], self.unit)
    def __str__(self):
        return str(self.Converse2Metres())
    def __repr__(self):
        return "Length(" + str(self.value) + ", '" + self.unit + "')"
```

Polymorphism : magic method

- เรียกใช้งาน

```
x = Length(4)
print(x)
z = Length(4.5, "yd") + Length(1)
print(repr(z))
print(z)
```

- ผลการทำงาน

```
4
Length(5.593613298337708, 'yd')
5.1148
```

Polymorphism : magic method

- ใน constructor จะรับมูลความยาวและหน่วย หากไม่กำหนดจะให้หน่วยเป็น m
- ฟังก์ชัน Converse2Metres ทำหน้าที่แปลงจากหน่วยอื่นๆ เป็นเมตร
- ฟังก์ชัน __add__ เป็น magic method ซึ่งจะเรียกใช้งานเมื่อกระทำ operator “+” ระหว่าง object ในคลาสเดียวกันและส่งกลับเป็น object
- ฟังก์ชัน __str__ เป็น magic method ซึ่งจะเรียกใช้งานเมื่อใช้กับคำสั่ง print หรือ str ทำหน้าที่แสดงข้อมูลที่เก็บที่แปลงเป็นเมตรแล้ว
- ฟังก์ชัน __repr__ เป็น magic method ซึ่งจะถูกเรียกใช้งานเมื่อใช้กับคำสั่ง repr หรือเมื่อเรียก method ใน interpreter ทำหน้าที่แสดงความยาวในหน่วยที่กำหนด

Polymorphism : magic method

- จากโปรแกรม เนื่องจาก method __add__ จะรับข้อมูลเป็น object จึงไม่สามารถไปบวกกับข้อมูล type อื่นได้ เช่น เขียนว่า

```
x = Length(1) +1
```

- ก็จะเกิด Error ดังนั้นควรแก้ไข __add__ ให้เป็นดังนี้ ก็จะทำงานได้ การทำงานคือ ตรวจสอบชนิดข้อมูลก่อน หากเป็นชนิดอื่นก็แปลงก่อนแล้วค่อยบวก

```
def __add__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit)
```

Polymorphism : magic method

- แต่ก็จะยังไม่สามารถทำงานในกรณีนี้ได้ เนื่องจาก ใช้กับ += ไม่ได้

```
x += Length(1)
```

- กรณีนี้ต้องใช้ __radd__


```
def __radd__(self, other):
    if type(other) == int or type(other) == float:
        l = self.Converse2Metres() + other
    else:
        l = self.Converse2Metres() + other.Converse2Metres()
    return Length(l / Length.__metric[self.unit], self.unit)
```

- แม้ในโปรแกรมจะเขียนเหมือนกัน แต่ผลการทำงานจะต่างกัน

Polymorphism : magic method

Binary Operators

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Extended Assignments

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

Polymorphism : magic method

Unary Operators

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Comparison Operators

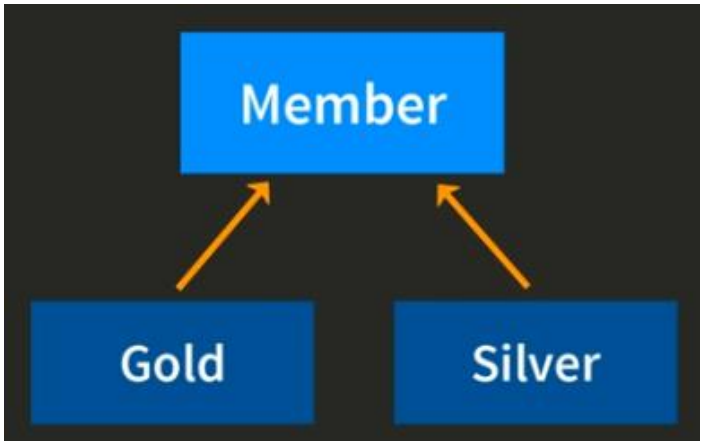
Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Polymorphism : magic method

- __getitem__ ใช้สำหรับดึงข้อมูลที่กำหนด
- __getslice__ ใช้สำหรับดึงข้อมูลในช่วงที่กำหนด
- __contains__ ใช้สำหรับ operator “in”
- https://rszalski.github.io/magicmethods/
- https://www.analyticsvidhya.com/blog/2021/08/explore-the-magic-methods-in-python/

Abstract Base Class

- Abstract base class อาจเรียกว่า **คลาสต้นแบบ** เพราะเป็นคลาสที่ไม่ใช้สำหรับสร้าง Instance โดยตรง แต่ทำหน้าที่เป็นโครงสำหรับคลาสที่สืบทอดไป เพื่อให้คลาสที่สืบทอดไป ต้อง implement ตามที่กำหนด
- ยกตัวอย่างระบบสมาชิก แบ่งเป็นสมาชิก 2 ประเภท คือ Gold กับ Silver โดยกำหนดว่าในคลาสที่สืบทอดไปจะต้อง implement เรื่องส่วนลด



Abstract Base Class

```python
from abc import ABC, abstractmethod

class Member(ABC):
    def __init__(self, m_id, fname, lname):
        self.m_id = m_id
        self.fname = fname
        self.lname = lname

    @abstractmethod
    def discount(self):
        pass

    def full_name(self):
        return "{} {}".format(self.fname, self.lname)

class Gold(Member):
    def discount(self):
        return .10

class Silver(Member):
    def discount(self):
        return .05
```

Abstract Base Class

- ในการใช้ abstract base class จะต้อง import จาก library ชื่อ abc
- คลาสใดที่ inherit จาก ABC จะไม่สามารถสร้าง instance ของคลาสนั้นได้ หากพยายามสร้าง instance จะเกิด Type Error
- Decorator @abstracmethod หากใช้กับ method ใด จะเป็นการบังคับว่าคลาสที่ inherit จากคลาสนั้นไป จะต้อง implement method นั้นใหม่เสมอ
- จะเห็นได้ว่าคลาส Gold และ Silver ที่ inherit มาจากคลาส Member ก็จะถูกบังคับให้ implement method ชื่อ discount
- ทั้งนี้เพื่อให้เกิด interface ที่เหมือนกันหมดสำหรับทุกคลาสที่ inherit มาจากคลาสที่เป็น abstract base class

Abstract Base Class

- นอกเหนือจาก abstract method แล้วยังมี abstract property อีกด้วย
- Class MediaLoader จะทำหน้าที่เป็น base class โดยบังคับให้ต้องกำหนด attribute ext และ method play

```
from abc import ABC, abstractmethod, abstractproperty

class MediaLoader(ABC):
    @abstractmethod
    def play(self):
        pass

    @abstractproperty
    def ext(self):
        pass
```

Abstract Base Class

- จะเห็นว่าเมื่อคลาส Ogg ทำการ Inherit มาจากคลาส MediaLoader
- จะต้องมีการกำหนด attribute ext และ method play มิฉะนั้นจะ Error

```
class Ogg(MediaLoader):
    ext = '.ogg'

    def play(self):
        pass

o = Ogg()
```

Inheritance vs Composition

- ในการออกแบบคลาส มีหลักการอยู่ข้อหนึ่งว่าควรออกแบบคลาสให้รับผิดชอบเพียงอย่างใดอย่างหนึ่ง จะขอยกตัวอย่าง
- สมมติว่า ในองค์กรแห่งหนึ่ง มีการจ้างพนักงาน 3 ประเภท
  - พนักงานประเภทที่ 1 รายชั่วโมง จะได้ค่าจ้างต่อ 1 ชั่วโมง + 1000
  - พนักงานประเภทที่ 2 รายเดือน จะได้เงินเดือน และ ค่า commission ต่อสัญญาที่หาได้
  - พนักงานประเภทที่ 3 Freelance จะได้ค่าจ้างรายชั่วโมง และ ค่า commission ต่อสัญญาที่หาได้

Inheritance vs Composition

- คลาส พนักงานรายชั่วโมง

```python
@dataclass
class HourlyEmployee:
    """Employee that's paid based on number of worked hours."""

    name: str
    id: int
    commission: float = 100
    contracts_landed: float = 0
    pay_rate: float = 0
    hours_worked: int = 0
    employer_cost: float = 1000

    def compute_pay(self) -> float:
        """Compute how much the employee should be paid."""
        return (
            self.pay_rate * self.hours_worked + self.employer_cost
            + self.commission * self.contracts_landed)
```

Inheritance vs Composition

- คลาส พนักงานรายเดือน

```
@dataclass
class SalariedEmployee:
    """Employee that's paid based on a fixed monthly salary."""

    name: str
    id: int
    commission: float = 100
    contracts_landed: float = 0
    monthly_salary: float = 0
    percentage: float = 1

    def compute_pay(self) -> float:
        """Compute how much the employee should be paid."""
        return (
            self.monthly_salary * self.percentage
            + self.commission * self.contracts_landed)
```

Inheritance vs Composition

- คลาส พนักงาน Freelance

```python
@dataclass
class Freelancer:
    """Freelancer that's paid based on number of worked hours."""

    name: str
    id: int
    commission: float = 100
    contracts_landed: float = 0
    pay_rate: float = 0
    hours_worked: int = 0

    def compute_pay(self) -> float:
        """Compute how much the employee should be paid."""
        return (
            self.pay_rate * self.hours_worked +
            self.commission * self.contracts_landed )
```

Inheritance vs Composition

- จะเห็นว่าทั้ง 3 คลาส มีปัญหา 2 ประการ
- ประการที่ 1 คือ ทั้ง 3 คลาสมี code ที่ซ้ำซ้อนกัน
  - พนักงาน รายเดือน และ Freelance ต่างก็มีค่า commission เหมือนกัน
  - พนักงาน รายชั่วโมง และ Freelance ต่างก็มีค่าจ้างรายชั่วโมงเหมือนกัน
- ประการที่ 2 คือ ทั้ง 3 คลาส ต่างก็รับผิดชอบกับข้อมูลหลายอย่าง
  - ข้อมูล พนักงาน ประกอบด้วย id และ name
  - ข้อมูล commission และ จำนวนสัญญาที่ทำได้
  - ข้อมูลชั่วโมง และ ค่าจ้างรายชั่วโมง
  - ข้อมูลค่าจ้างรายเดือน

Inheritance vs Composition

- การแก้ไขปัญหาวิธีแรก คือ ใช้ Inheritance โดยจะสร้าง abstract base class ชื่อ Employee สำหรับเก็บ id และ name โดยมี compute_pay เป็น method บังคับ

```python
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class Employee(ABC):
    """Basic representation of an employee at the company."""

    name: str
    id: int

    @abstractmethod
    def compute_pay(self) -> float:
        """Compute how much the employee should be paid."""
```

Inheritance vs Composition

- จากนั้นจะสร้างคลาส HourlyEmployee และ SaraliedEmployee ที่ inherit มา

```python
@dataclass
class HourlyEmployee(Employee):
    """Employee that's paid based on number of worked hours."""

    pay_rate: float
    hours_worked: int = 0
    employer_cost: float = 1000

    def compute_pay(self) -> float:
        return self.pay_rate * self.hours_worked + self.employer_cost

@dataclass
class SalariedEmployee(Employee):
    """Employee that's paid based on a fixed monthly salary."""

    monthly_salary: float
    percentage: float = 1

    def compute_pay(self) -> float:
        return self.monthly_salary * self.percentage
```

Inheritance vs Composition

- และคลาส Freelancer ที่ Inherit มาจากคลาส Employee เช่นกัน
- ทั้ง 3 คลาส จะมี method compute_pay ที่เขียนต่างกัน
- ทั้ง 3 คลาส จะยังไม่มีข้อมูลเรื่อง commission ดังนั้นจะต้อง inherit ทั้ง 3 คลาส เพื่อสร้าง พนักงานที่ได้ commission ต่อไปอีก

```python
@dataclass
class Freelancer(Employee):
    """Freelancer that's paid based on number of worked hours."""

    pay_rate: float
    hours_worked: int = 0
    vat_number: str = ""

    def compute_pay(self) -> float:
        return self.pay_rate * self.hours_worked
```

Inheritance vs Composition

```python
@dataclass
class SalariedEmployeeWithCommission(SalariedEmployee):
    """Employee that's paid based on a fixed monthly salary and that gets a commission."""

    commission: float = 100
    contracts_landed: float = 0

    def compute_pay(self) -> float:
        return super().compute_pay() + self.commission * self.contracts_landed


@dataclass
class HourlyEmployeeWithCommission(HourlyEmployee):
    """Employee that's paid based on number of worked hours and that gets a commission."""

    commission: float = 100
    contracts_landed: float = 0

    def compute_pay(self) -> float:
        return super().compute_pay() + self.commission * self.contracts_landed
```

Inheritance vs Composition

- ทั้ง 3 คลาส เป็นคลาสสำหรับพนักงานทั้ง 3 ประเภทที่มีค่า commission ด้วย
- จะเห็นว่าปัญหาทั้ง 2 เรื่องที่กล่าวมาก่อนหน้านี้ไม่มีแล้ว
- Code มีส่วนที่ซ้ำซ้อนกันน้อยลงแล้ว แต่ compute_pay ยังมีส่วนที่ซ้ำอยู่
- แต่ละคลาสรับผิดชอบข้อมูลเรื่องเดียว Employee รับผิดชอบข้อมูลพนักงาน และคลาส inherit ชั้นที่ 1 รับผิดชอบคำนวณค่าจ้าง และคลาส inherit ชั้นที่ 2 รับผิดชอบเรื่องค่า commission แต่จะเห็นว่าใช้คลาสค่อนข้างมาก (ถ้ามีอีก 1 ประเด็น???)

```python
@dataclass
class FreelancerWithCommission(Freelancer):
    """Freelancer that's paid based on number of worked hours and that gets a commission."""

    commission: float = 100
    contracts_landed: float = 0

    def compute_pay(self) -> float:
        return super().compute_pay() + self.commission * self.contracts_landed
```

Inheritance vs Composition

- จะเห็นได้ว่ากรณีที่มีข้อมูลที่สัมพันธ์กันหลายส่วน การใช้ inheritance อาจไม่ใช่ทางเลือกที่ดีที่สุด
- กรณีนี้แนะนำให้แยกข้อมูล Contract และ Commission ออกมาเป็นคลาส

```
class Contract(ABC):
    """Represents a contract and a payment process for a particular employeee."""

    @abstractmethod
    def get_payment(self) -> float:
        """Compute how much to pay an employee under this contract."""

@dataclass
class Commission(ABC):
    """Represents a commission payment process."""

    @abstractmethod
    def get_payment(self) -> float:
        """Returns the commission to be paid out."""
```

Inheritance vs Composition

- จากนั้นจะ inherit คลาส Commission เป็น ContractCommission ทำหน้าที่คำนวณค่า commission ต่อจำนวนสัญญาที่ทำได้

```python
@dataclass
class ContractCommission(Commission):
    """Represents a commission payment process based on the number of contracts landed."""

    commission: float = 100
    contracts_landed: int = 0

    def get_payment(self) -> float:
        """Returns the commission to be paid out."""
        return self.commission * self.contracts_landed
```

Inheritance vs Composition

- จากนั้นจึงนำ Contract และ Commission มาเป็น Composition ของ Employee

```python
@dataclass
class Employee:
    """Basic representation of an employee at the company."""

    name: str
    id: int
    contract: Contract
    commission: Optional[Commission] = None

    def compute_pay(self) -> float:
        """Compute how much the employee should be paid."""
        payout = self.contract.get_payment()
        if self.commission is not None:
            payout += self.commission.get_payment()
        return payout
```

Inheritance vs Composition

- จึงค่อยสร้างคลาสของ contract รายชั่วโมง และ contract รายเดือน

```python
@dataclass
class HourlyContract(Contract):
    """Contract type for an employee being paid on an hourly basis."""

    pay_rate: float
    hours_worked: int = 0
    employer_cost: float = 1000

    def get_payment(self) -> float:
        return self.pay_rate * self.hours_worked + self.employer_cost

@dataclass
class SalariedContract(Contract):
    """Contract type for an employee being paid a monthly salary."""

    monthly_salary: float
    percentage: float = 1

    def get_payment(self) -> float:
        return self.monthly_salary * self.percentage
```

Inheritance vs Composition

- จะเห็นว่าความซ้ำซ้อนของ code ไม่มีแล้ว และ แต่ละคลาสรับผิดชอบเพียงเรื่องเดียว และความยาวของ code ก็ใกล้เคียงกัน
- แต่หากเพิ่มข้อมูล เช่น โบนัสรายปี การใช้ composition จะแก้ไขน้อยกว่า
- ดังนั้น กรณีที่ข้อมูลมีหลายส่วนเกี่ยวข้องกัน ควรพิจารณาใช้ composition

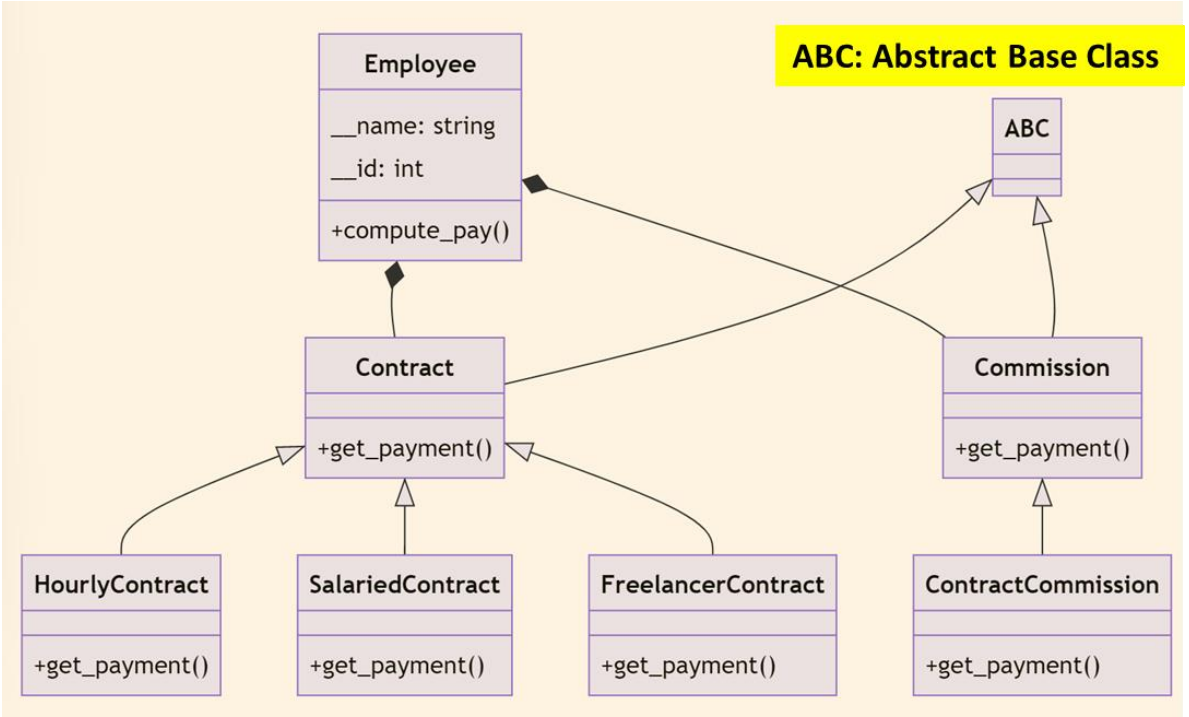
```python
@dataclass
class FreelancerContract(Contract):
    """Contract type for a freelancer (paid on an hourly basis)."""

    pay_rate: float
    hours_worked: int = 0
    vat_number: str = ""

    def get_payment(self) -> float:
        return self.pay_rate * self.hours_worked
```

Inheritance vs Composition

```python
def main() -> None:
    """Main function."""

    henry_contract = HourlyContract(pay_rate=50, hours_worked=100)
    henry = Employee(name="Henry", id=12346, contract=henry_contract)
    print(
        f"{henry.name} worked for {henry_contract.hours_worked} hours "
        f"and earned ${henry.compute_pay()}."
    )

    sarah_contract = SalariedContract(monthly_salary=5000)
    sarah_commission = ContractCommission(contracts_landed=10)
    sarah = Employee(
        name="Sarah", id=47832, contract=sarah_contract, commission=sarah_commission
    )
    print(
        f"{sarah.name} landed {sarah_commission.contracts_landed} contracts "
        f"and earned ${sarah.compute_pay()}."
    )
```

Inheritance vs Composition

- ไม่ว่าจะใช้ inheritance หรือ composition ก็ตาม สิ่งสำคัญ คือ การใช้ interface แบบเดียวกัน
- เพราะจะทำให้โปรแกรมที่เรียกใช้ interface ไม่ต้องแยกการเรียกใช้ออกไป สามารถใช้การเรียกใช้ แบบเดียวกันได้



get_payment()

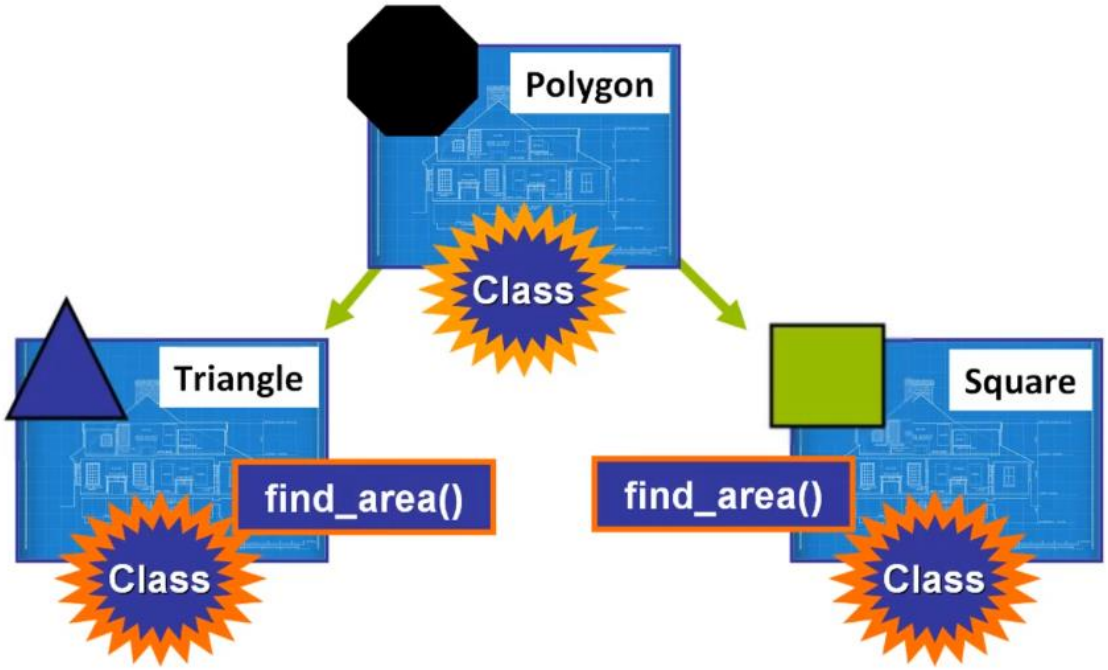
Inheritance

- การเรียนที่ผ่านมาได้กล่าวถึง Inheritance ไปบ้างแล้ว สำหรับภาษา Python จะมีความสามารถในการ “สืบทอด” จากหลายคลาส ซึ่งในบางภาษาไม่มี
- คลาส Button inherit มากจาก 2 คลาส จึงมีความสามารถของทั้ง 2 คลาส

```
class Rectangle:
    def __init__(self, length, width, color):
        self.length = length
        self.width = width
        self.color = color

class GUIElement:
    def click(self):
        print("The object was clicked...")

class Button(Rectangle, GUIElement):
    def __init__(self, length, width, color, text):
        Rectangle.__init__(self, length, width, color)
        self.text = text

bt = Button(10, 20, "RED", "Confirm")
bt.click()
```

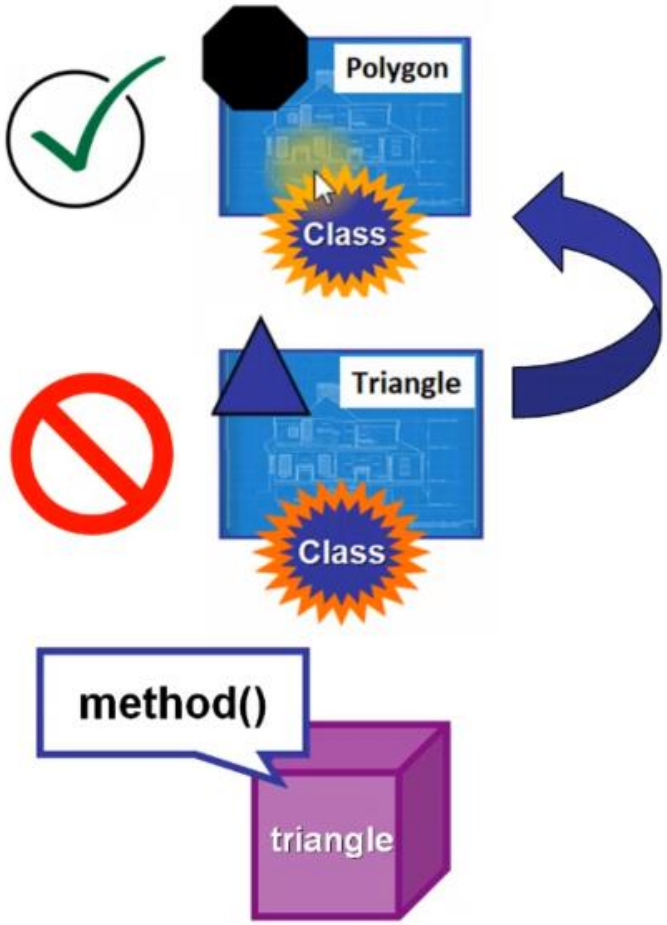
Inheritance

- ในเรื่อง polymorphism ได้กล่าวถึง class ที่มี interface เหมือนกัน ซึ่งถือได้ว่าเป็นจุดที่สำคัญของ OOP เนื่องจากทำให้โปรแกรมที่เขียนขึ้น สามารถเรียกใช้แบบเดียวกันกับข้อมูลที่ต่างกัน (เช่น การจ่ายเงินแบบต่างๆ)
- ตัวอย่างนี้ แสดง method find_area() ซึ่งเป็น interface ชื่อเดียว แต่ทำงานต่างกัน



Inheritance

- ดังนั้นในแต่ละคลาส ก็อาจจะมี method ชื่อเดียวกันได้ คำถาม คือ หากมีการเรียกใช้ method จะเรียก method นั้นจากคลาสใด
- การค้นหา Method จะใช้หลักการตามรูป คือ จะค้นจากจากคลาสลำดับขึ้นไป 1 ชั้น หากไม่พบ จึงจะหาใน Class ลำดับเหนือขึ้นไปเรื่อยๆ



Inheritance

- นอกจากนั้น กรณีที่ superclass กำหนด method เอาไว้ ใน subclass สามารถกำหนด method ชื่อเดียวกันซ้ำได้ โดยเรียกว่า method overloading

```
class Teacher:

    def __init__(self, full_name, teacher_id):
        self.full_name = full_name
        self.teacher_id = teacher_id

    def welcome_students(self):
        print(f"Welcome to class!, I'm your teacher. My name is {self.full_name}")

class ScienceTeacher(Teacher):

    def welcome_students(self):
        print(f"Science is amazing.")
        print(f"Welcome to class. I'm your teacher: {self.full_name}")
```

Inheritance

- จากตัวอย่างก่อนหน้าจะเห็นว่ามี code ที่ซ้ำกันบางส่วน ดังนั้นหากจะไม่ให้ซ้ำจะต้องให้ method welcome_students ในคลาส ScienceTeccher ไปเรียกใช้ method welcome_students ในคลาส Teacher ตามรูป (จะใช้ชื่อคลาสก็ได้)

```
class Teacher:

    def __init__(self, full_name, teacher_id):
        self.full_name = full_name
        self.teacher_id = teacher_id

    def welcome_students(self):
        print(f"Welcome to class!, I'm your teacher. My name is {self.full_name}")

class ScienceTeacher(Teacher):

    def welcome_students(self):
        print("Science is amazing.")
        super().welcome_students()
```

Inheritance

- สามารถ Inherit จากคลาสมาตรฐานก็ได้ เช่น เพิ่มการ search ให้ list

```
class ContactList(list):
    def search(self, name):
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

class Contact:
    all_contacts = ContactList()

    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)
```

Inheritance

```
1  class A:
2
3      def x(self):
4          print("Class A")
5
6  class B(A):
7
8      def x(self):
9          print("Class B")
10
11 a = A()
12 b = B()
13
14 # Output?
15 a.x()
16 b.x()
```

- Code นี้จะแสดงอะไร

○
```
1  Class A
2  Class A
```

○
```
1  Class A
2  Class B
```

○
```
1  Class B
2  Class B
```

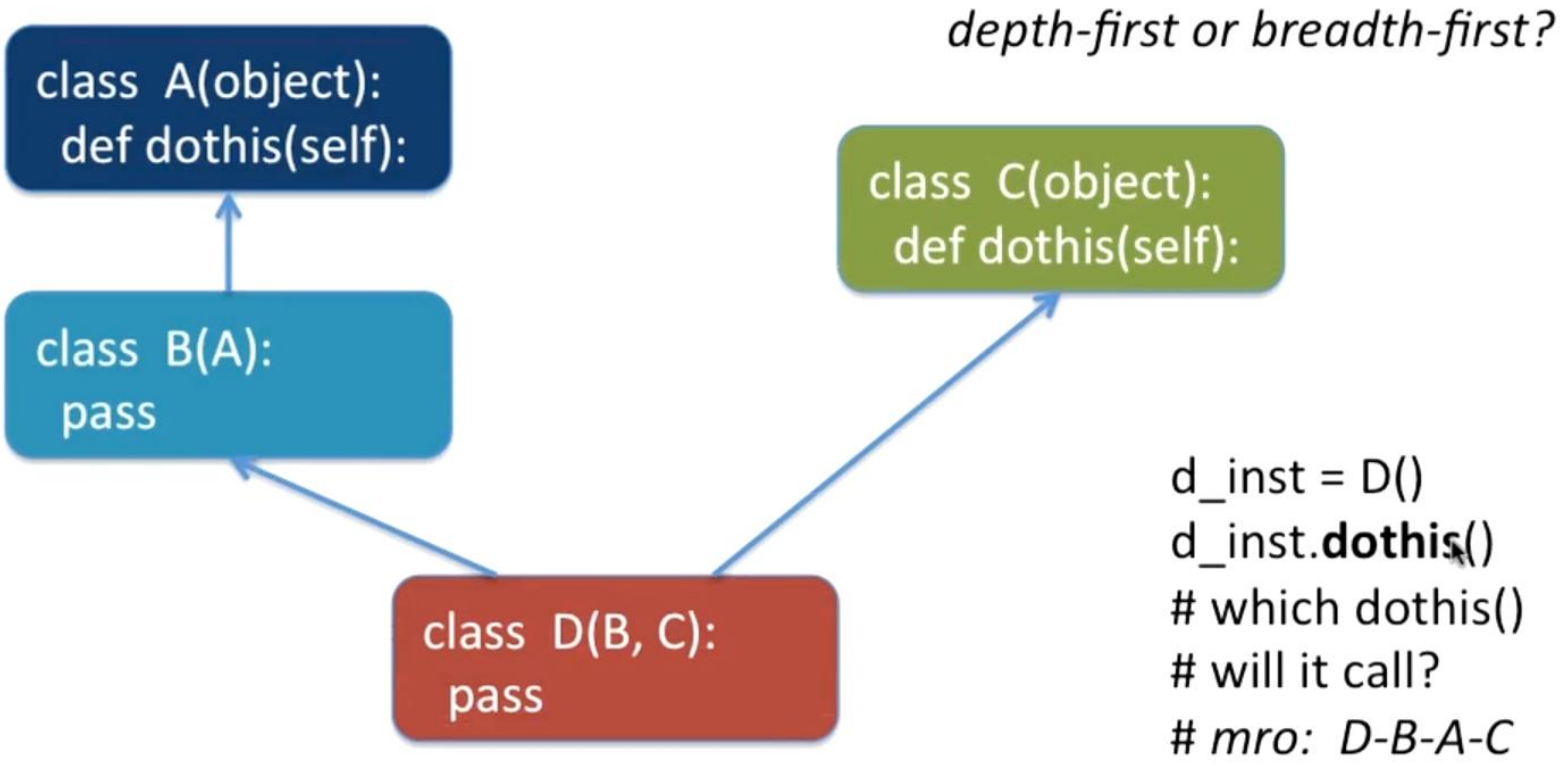
Inheritance

- ในกรณีที่ method ชื่อซ้ำกันระหว่าง superclass กับ subclass
- คำถาม คือ จะเรียกใช้ method จากคลาสใด ปัญหานี้เรียกว่า method resolution order (MRO) ซึ่งหากมีโครงสร้างคลาสไม่ซับซ้อน ก็ไม่มีปัญหาอะไร

```
1  # Python program showing
2  # how MRO works
3
4  class A:
5      def rk(self):
6          print(" In class A")
7  class B(A):
8      def rk(self):
9          print(" In class B")
10
11 r = B()
12 r.rk()
13
```

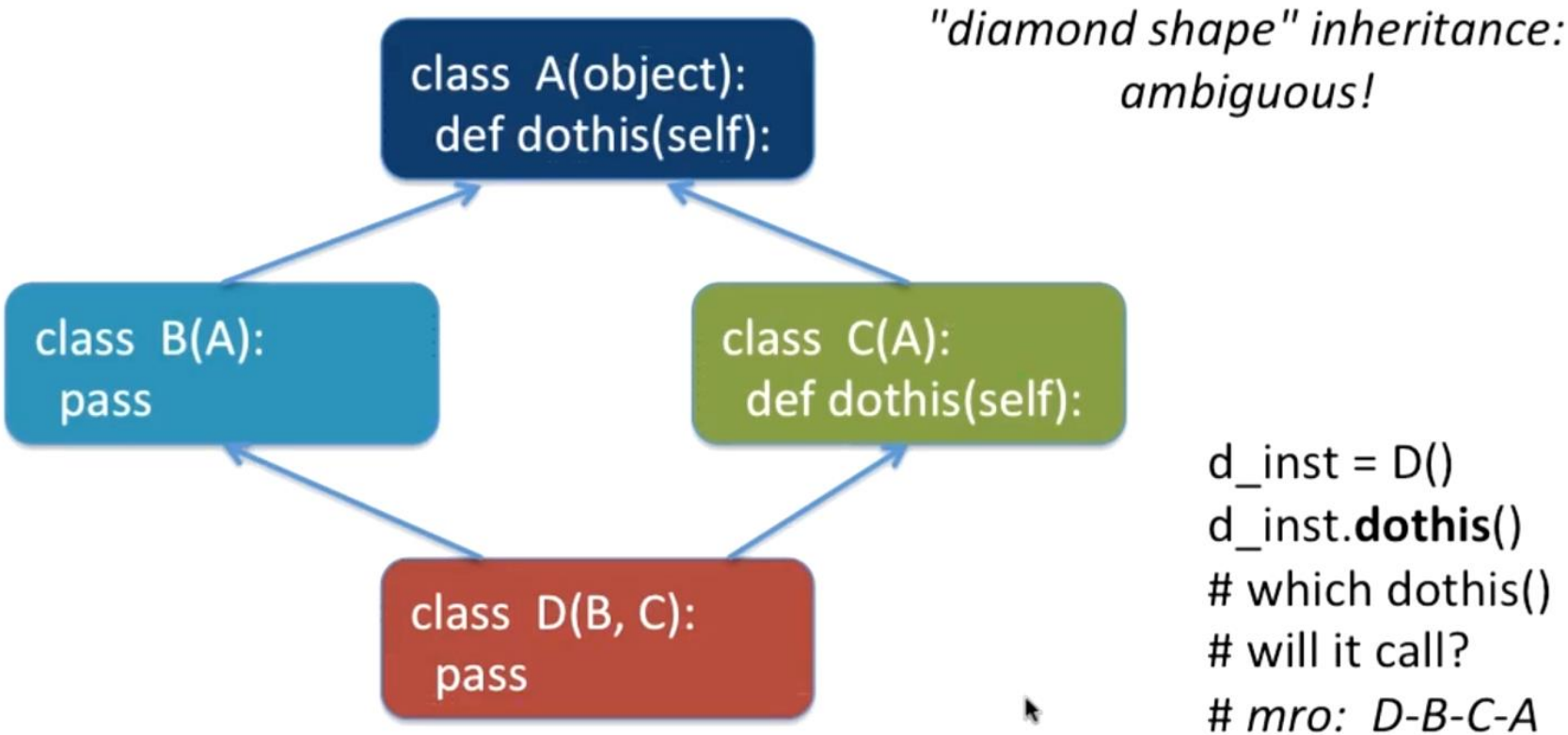
Inheritance

- แต่ในกรณีที่โครงสร้างคลาสซับซ้อนขึ้น เช่น จากรูปจะมีวิธีการหาอย่างไร
- กรณีนี้ python จะใช้วิธีที่เรียกว่า depth-first search คือ หาลงทางลึกก่อน กรณีนี้มีการอ้างถึงคลาส B ก่อน ดังนั้นจะค้นหาจาก D -> B -> A -> C



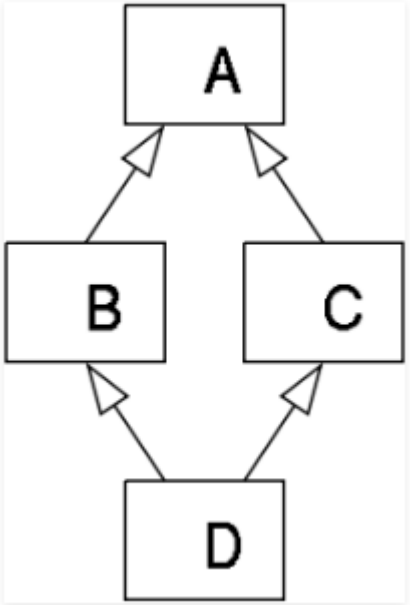
Inheritance

- มีปัญหาหนึ่งในการทำ MRO ของ python ปัญหานี้เรียกว่า diamond problem โดยได้ชื่อมาจากโครงสร้างคลาสที่เป็น diamond shape โดยกรณีที่คลาส inherit มาจาก class เดียวกัน จะไม่ใช้วิธี depth-first search แต่จะค้นหาตามลำดับในการอ้างคลาส



Inheritance

- เช่นจาก code นี้ จะแสดงผลอะไร
- ถ้าลบ rk ในคลาส b ออก จะเป็นอย่างไร




```
class A:
    def rk(self):
        print("In class A")

class B:
    def rk(self):
        print("In class B")

class C:
    def rk(self):
        print("In class C")

class D(B, C):
    pass

r = D()
r.rk()
```

Inheritance

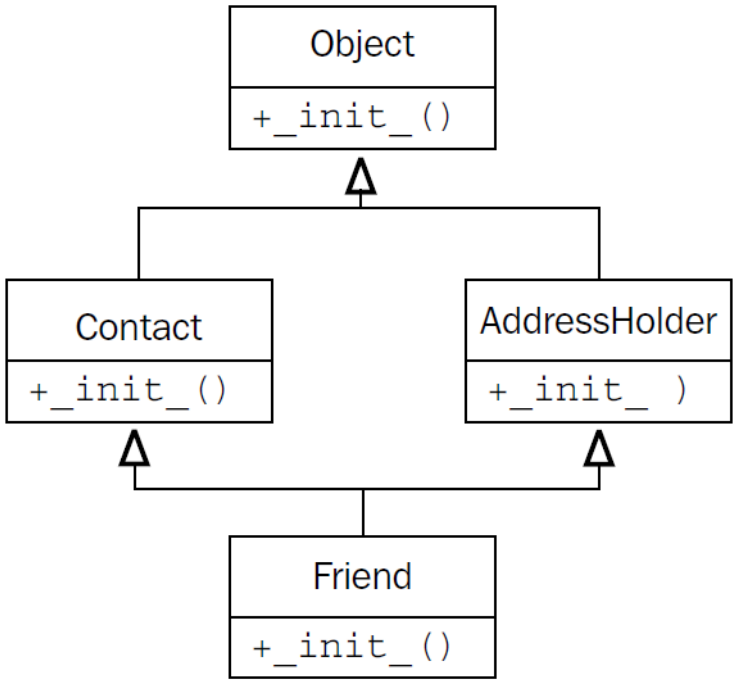
- เราสามารถตรวจสอบ MRO ได้

```
# Python program to show the order
# in which methods are resolved

class A:
    def rk(self):
        print(" In class A")
class B:
    def rk(self):
        print(" In class B")

# classes ordering
class C(A, B):
    def __init__(self):
        print("Constructor C")

r = C()

# it prints the lookup order
print(C.__mro__)
print(C.mro())
```

```
Constructor C
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>)
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

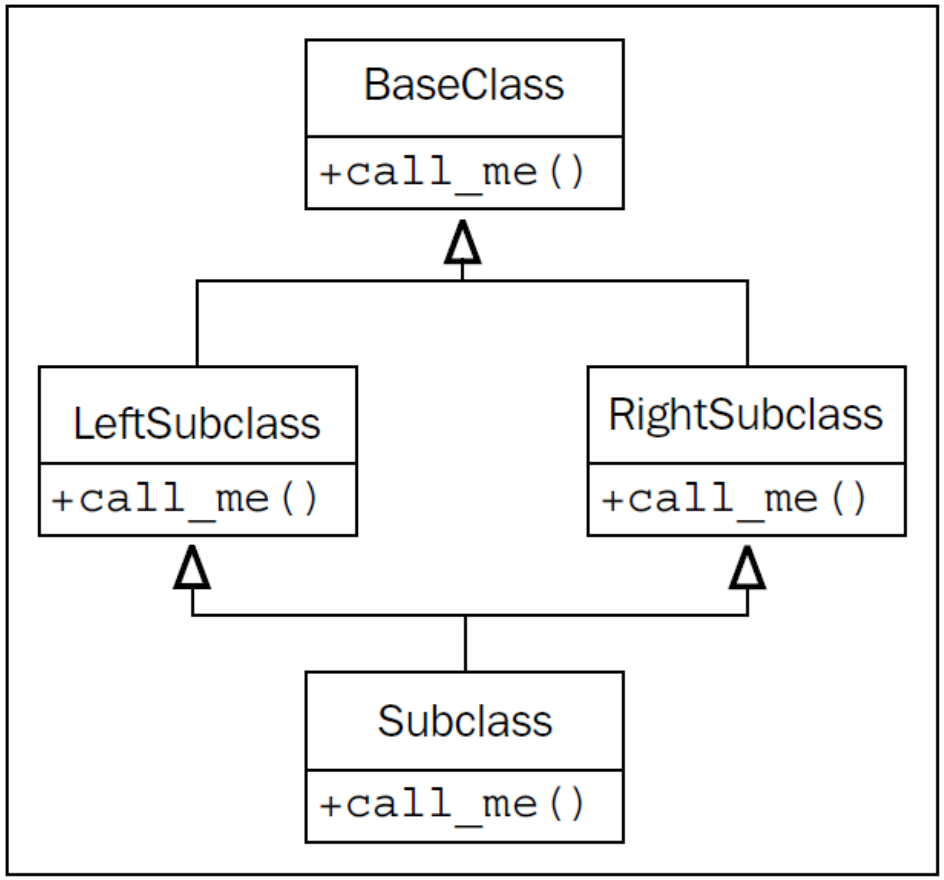
Inheritance

- ลองดูอีกตัวอย่าง จากรูปทุกคลาส inherit มาจากคลาส Object ซึ่งแปลว่าในการทำงาน top superclass อาจจะรัน method __init__ 2 ครั้งได้ หากใน subclass มีการเรียก constructor ของ superclass โดยใช้ชื่อคลาส



Inheritance

- จะจำลองเหตุการณ์โดยสร้าง method ชื่อ call_me ในทุก class ตามรูป



Inheritance

- จากนั้นให้มีการเรียก call_me จากระดับล่างขึ้นมาทุกระดับ

```
class BaseClass:
    num_base_calls = 0

    def call_me(self):
        print("Calling method on Base Class")
        self.num_base_calls += 1

class LeftSubclass(BaseClass):
    num_left_calls = 0

    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Left Subclass")
        self.num_left_calls += 1
```

Inheritance

```
class RightSubclass(BaseClass):
    num_right_calls = 0

    def call_me(self):
        BaseClass.call_me(self)
        print("Calling method on Right Subclass")
        self.num_right_calls += 1

class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0

    def call_me(self):
        LeftSubclass.call_me(self)
        RightSubclass.call_me(self)
        print("Calling method on Subclass")
        self.num_sub_calls += 1
```

Inheritance

- จากนั้นเรียกใช้ดังนี้

```
s = Subclass()
s.call_me()
print(s.num_sub_calls, s.num_left_calls,
      s.num_right_calls, s.num_base_calls)
```

- จะมีการทำงานลำดับอย่างไร

```
Calling method on Base Class
Calling method on Left Subclass
Calling method on Base Class
Calling method on Right Subclass
Calling method on Subclass
1 1 1 2
```

- จะเห็นว่ามีการเรียก base class 2 ครั้ง ซึ่งต้องระวังในกรณีนี้

Inheritance

- แต่หากใช้ super() แทน

```
class BaseClass:
    num_base_calls = 0

    def call_me(self):
        print("Calling method on Base Class")
        self.num_base_calls += 1

class LeftSubclass(BaseClass):
    num_left_calls = 0

    def call_me(self):
        super().call_me()
        print("Calling method on Left Subclass")
        self.num_left_calls += 1
```

Inheritance

```
class RightSubclass(BaseClass):
    num_right_calls = 0

    def call_me(self):
        super().call_me()
        print("Calling method on Right Subclass")
        self.num_right_calls += 1

class Subclass(LeftSubclass, RightSubclass):
    num_sub_calls = 0

    def call_me(self):
        super().call_me()
        print("Calling method on Subclass")
        self.num_sub_calls += 1
```

Inheritance

- เมื่อเรียกทำงาน จะเห็นว่ามีการใช้ base class เพียงครั้งเดียว
- ลำดับการเรียกใช้จะเห็นว่า มีการเรียกใช้จาก Subclass ไปยัง Left Subclass ไปยัง Right Subclass แล้วจึงไปยัง Base Class ทั้งที่ Base Class เป็น super() ของ Left Subclass
- นี่เป็นความแตกต่างระหว่างการใช้ super() กับการเรียกโดยใช้ชื่อคลาส

```
Calling method on Base Class
Calling method on Right Subclass
Calling method on Left Subclass
Calling method on Subclass
1 1 1 1
```

Inheritance : Difference set of argument

- ปัญหาหนึ่งที่อาจเกิดขึ้นกับการ inherit หลายๆ ชั้น คือ พารามิเตอร์ที่ส่งไปที่ instance ของคลาสลำดับสุดท้าย บางส่วนอาจใช้โดยคลาสลำดับที่สูงกว่า เช่น มีคลาสดังนี้

```
class Contact:
    all_contacts = []

    def __init__(self, name, email):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

class AddressHolder:
    def __init__(self, street, city, state, code):
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

Inheritance : Difference set of argument

- สมมติว่าสร้างคลาส

```
class Friend(Contact, AddressHolder):
    def __init__(self, name, email, phone,street, city, state, code):
        Contact.__init__(self, name, email)
        AddressHolder.__init__(self, street, city, state, code)
        self.phone = phone
```

- จะเห็นว่าคลาสที่ inherit มา มีพารามิเตอร์ไม่ซ้ำกันเลย เมื่อเรียก constructor ของ superclass แบบระบุชื่อ จึงไม่มีปัญหา แต่ถ้าเปลี่ยนเป็น super() จะมีปัญหาหรือไม่

```
class Friend(Contact, AddressHolder):
    def __init__(self, name, email, phone,street, city, state, code):
        super().__init__(self, name, email)
        super().__init__(self, street, city, state, code)
        self.phone = phone

f = Friend("Terry", "terry@abc.co", "081-234-5678", \
           "sukumvit", "bkk", "bkk", 10520)
```

Inheritance : Difference set of argument

- โปรแกรมจาก slide ที่แล้ว เมื่อรันจะพบ Error เนื่องจากจะฟ้องว่า

```
Traceback (most recent call last):
  File "main.py", line 22, in <module>
    f = Friend("Terry", "terry@abc.co", "081-234-5678", \
  File "main.py", line 18, in __init__
    super().__init__(self, name, email)
TypeError: __init__() takes 3 positional arguments but 4 were given
```

- วิธีการแก้ไขในกรณีนี้ ต้องใช้วิธีการส่ง argument เป็น List หรือ Dictionary แล้วให้แต่ละคลาสดึงข้อมูลที่ต้องการใช้งานไปใช้เอง
- ถ้าส่งเป็น argument list ให้ใช้เป็น *args โดยข้อมูลจะต้องเรียงตามลำดับ
- ถ้าส่งเป็น argument dictionary จะใช้เป็น **kwargs โดย key จะเป็นชื่อของ argument และ value เป็นค่าของ argument จึงควรใช้แบบนี้จะดีกว่า

Inheritance : Difference set of argument

- ดังนั้นจะปรับปรุงโปรแกรมเป็นดังนี้ โดยให้ระบุเฉพาะ argument ที่ใช้ในแต่ละคลาส ดังนั้น argument อื่นๆ ก็จะอยู่ใน **kwargs

```
class Contact:
    all_contacts = []

    def __init__(self, name='', email='', **kwargs):
        self.name = name
        self.email = email
        Contact.all_contacts.append(self)

class AddressHolder:
    def __init__(self, street='', city='', state='', code='', **kwargs):
        self.street = street
        self.city = city
        self.state = state
        self.code = code
```

Inheritance : Difference set of argument

- ให้สังเกตุว่าจะใช้ super().__init__(**kwargs) เพียงครั้งเดียว โดยจะเกิดผลเป็นการเรียกใช้ superclass ทั้ง 2 คลาส

```
class Friend(Contact, AddressHolder):
    def __init__(self, phone='',**kwargs):
        super().__init__(**kwargs)
        self.phone = phone


f = Friend(name="Terry", email="terry@abc.co", phone="081-234-5678", \
           street="sukumvit", city="bkk", state="bkk", code=10520)
```

Inheritance : Difference set of argument

- วิธีการข้างต้นจะมีปัญหา 1 จุด คือ ใน argument list จะไม่มี phone ดังนั้น หากใน superclass ใดมีการใช้ phone จะมีปัญหา
- วิธีแก้ 1 : ให้รวม phone ใน **kwargs และให้ class Friend ค้นหาใน dictionary : kwargs[‘phone’]
- วิธีแก้ 2 : เพิ่ม phone เข้าไปภายหลัง kwargs[‘phone’] = phone
- วิธีแก้ 3 : เพิ่ม phone เข้าไปโดยใช้ kwargs.update
- วิธีแก้ 4 : ส่ง phone ใน __init__(phone=phone, **kwargs)

Mixin classes in Python

- ใน OOP จะมีวิธีการในการเพิ่มหรือเปลี่ยนแปลงพฤติกรรมของคลาสอื่น โดยไม่ใช้วิธี inherit เรียกว่า mixin สมมติว่ามีคลาส Vehicle และมี subclass Car และ Boat

```
class Vehicle:
    """A generic vehicle class."""

    def __init__(self, position):
        self.position = position

    def calculate_route(depart, to):
        pass

   def travel(self, destination):
        route = calculate_route(self.position, to)
        self.move_along(route)

class Car(Vehicle):
    pass

class Boat(Vehicle):
    pass
```

Mixin classes in Python

- หากต้องการให้รถยนต์มีคุณสมบัติ คือ มีวิทยุ แต่ไม่ต้องการให้เรือมีวิทยุด้วย สามารถจะสร้างคลาส Mixin ที่เป็นวิทยุ เพื่อนำมาให้คลาส Car inherit แล้วมีคุณสมบัติเพิ่มได้

```
class RadioUserMixin(object):
    def __init__(self):
        self.radio = Radio()

    def play_song_on_station(self, station):
        self.radio.set_station(station)
        self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
    pass
```

Mixin classes in Python

- ดูอีกตัวอย่างหนึ่ง สมมติว่ามีคลาส Person และ Employee

```
class Person:
    def __init__(self, name):
        self.name = name

class Employee(Person):
    def __init__(self, name, skills, dependents):
        super().__init__(name)
        self.skills = skills
        self.dependents = dependents

if __name__ == '__main__':
    e = Employee(
        name='John',
        skills=['Python Programming''Project Management'],
        dependents={'wife': 'Jane', 'children': ['Alice', 'Bob']}
    )
```

Mixin classes in Python

- หากมีความต้องการให้สามารถ Export ข้อมูลใน object ออกมาเป็น dictionary ได้ด้วย สามารถสร้างเป็นคลาส DictMixin

```python
class DictMixin:
    def to_dict(self):
        return self._traverse_dict(self.__dict__)

    def _traverse_dict(self, attributes: dict) -> dict:
        result = {}
        for key, value in attributes.items():
            result[key] = self._traverse(key, value)

        return result

    def _traverse(self, key, value):
        if isinstance(value, DictMixin):
            return value.to_dict()
        elif isinstance(value, dict):
            return self._traverse_dict(value)
        elif isinstance(value, list):
            return [self._traverse(key, v) for v in value]
        elif hasattr(value, '__dict__'):
            return self._traverse_dict(value.__dict__)
        else:
            return value
```

Mixin classes in Python

- จากนั้นให้คลาส Customer inherit ก็จะสามารถพิมพ์เป็น dict ออกมาได้

```
class Employee(DictMixin, Person):
    def __init__(self, name, skills, dependents):
        super().__init__(name)
        self.skills = skills
        self.dependents = dependents

e = Employee(
    name='John',
    skills=['Python Programming', 'Project Management'],
    dependents={'wife': 'Jane', 'children': ['Alice', 'Bob']}
)

print(e.to_dict())
```

```
{'name': 'John', 'skills': ['Python Programming', 'Project Management'],
'dependents': {'wife': 'Jane', 'children': ['Alice', 'Bob']}}
```

Exception Handling

- กรณีที่มีความผิดพลาดในคลาส จะไม่นิยมให้แสดง Error โดยการ print
- แต่จะใช้กลไกของ Exception โดยกำหนดให้คลาส Raise exception ขึ้นมา

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient balance")
        else:
            self.balance -= amount
```

Exception Handling

- และในส่วนที่เรียกใช้ ให้ดำเนินการตามนี้

```
account = BankAccount(1000)
try:
    account.withdraw(1500)
except ValueError as e:
    print("Failed to withdraw:", str(e))
```

For your attention