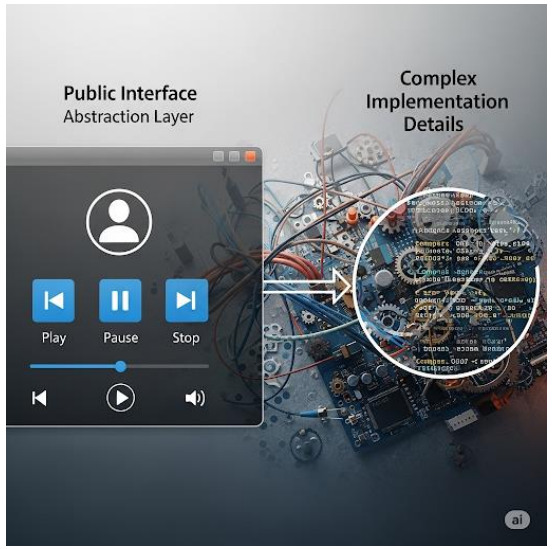
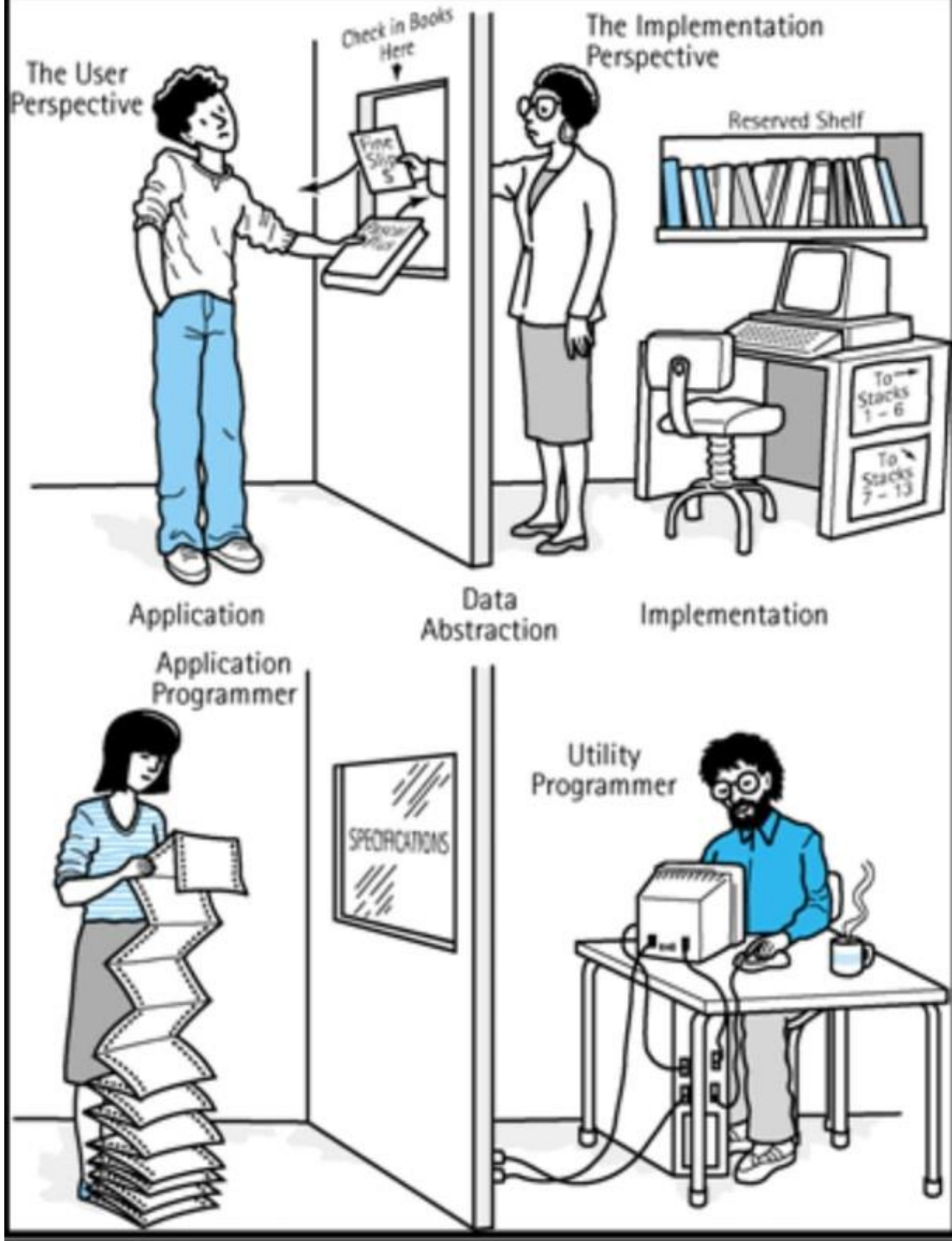
Object Oriented Programming in Python (Advanced)

SHUTCHON PREMCHAISAWATT

Abstraction (แนวคิดเชิงนามธรรม)

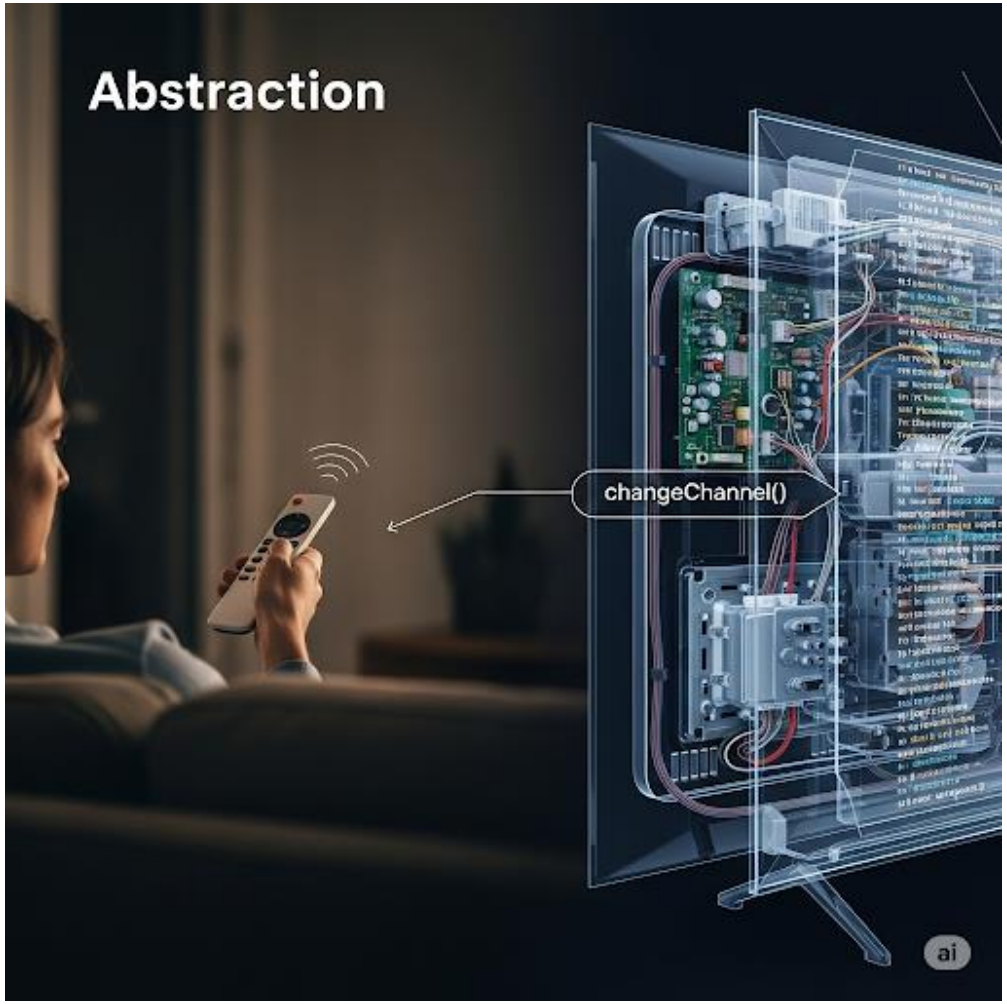
Abstraction คือการ "ซ่อนรายละเอียดที่ซับซ้อน และแสดงเฉพาะฟังก์ชันการทำงานที่จำเป็น" ค่ะ พูดอีกอย่างคือ การสร้าง "พิมพ์เขียวแม่แบบ" หรือ "โครงสร้างบังคับ" ขึ้นมา เพื่อกำหนดว่าคลาสอื่นๆ ที่จะเข้ามาเกี่ยวข้อง จะต้องทำอะไรได้บ้าง แต่เราจะไม่สนใจว่า มันจะทำสิ่งนั้นๆ ด้วยวิธีไหน





Abstraction

- ลองนึกถึงรีโมททีวีดูนะคะ 📺
- **สิ่งที่เราเห็น (Abstraction):** เราเห็นแค่ปุ่มเปิด-ปิด, ปุ่มเปลี่ยนช่อง, ปุ่มเพิ่ม-ลดเสียง เราแค่รู้ว่าถ้ากดปุ่มเปิด-ปิด ทีวีก็จะเปิดหรือดับ เราไม่จำเป็นต้องรู้เลยว่าพอเรากดปุ่มแล้ว มันส่งสัญญาณอินฟราเรดแบบไหน วงจรไฟฟ้าข้างในทำงานอย่างไร หรือทีวีถอดรหัสสัญญาณนั้นอย่างไร
- **สิ่งที่ถูกซ่อน (Implementation):** คือกลไกการทำงานที่ซับซ้อนทั้งหมดที่อยู่เบื้องหลังปุ่มเหล่านั้น
- Abstraction ใน OOP ก็ทำงานแบบเดียวกัน คือ การสร้าง "Interface" (เหมือนหน้ารีโมท) ที่บังคับว่าคลาสลูกที่จะสืบทอดไป *จะต้องมีเมธอดตามที่กำหนด* เพื่อรับประกันว่า Object ทุกชิ้นที่สร้างจากพิมพ์เขียวตระกูลนี้จะมีความสามารถพื้นฐานเหมือนกันแน่นอน



สร้าง Abstract

ใน Python เราจะใช้โมดูลที่มีชื่อว่า abc (Abstract Base Classes) เพื่อสร้างคลาสแม่แบบนี้ขึ้นมาค่ะลองดูตัวอย่าง "ยานพาหนะ" กันนะคะ เราจะสร้างคลาสแม่แบบชื่อ Vehicle ที่บังคับว่ายานพาหนะทุกชนิด จะต้อง start_engine() และ drive() ได้

มาดูโค้ดกันค่ะfrom abc import ABC, abstractmethod: เราต้อง import 2 อย่างนี้เข้ามาก่อนclass Vehicle(ABC):: การสืบทอดจาก ABC ทำให้คลาสนี้กลายเป็น Abstract Class@abstractmethod: นี่คือ "ตัวบังคับ" ค่ะ เมื่อเราแปะสิ่งนี้ไว้บนเมธอดไหน คลาสลูกทุกคลาสที่สืบทอดจาก Vehicle จะต้อง สร้างเมธอดชื่อเดียวกันนี้ขึ้นมา (ตรงนี้เรียกว่า Override) ไม่เช่นนั้น Python จะฟ้อง Error และเราจะไม่สามารถสร้าง Object จากคลาสลูกนั้นได้เลย! pass: ในเมธอดของคลาสแม่แบบ เรามักจะใส่แค่ pass เพราะเราไม่สนใจวิธีการทำงานของมัน เราแค่ "จองชื่อ" เมธอดไว้เฉยๆ

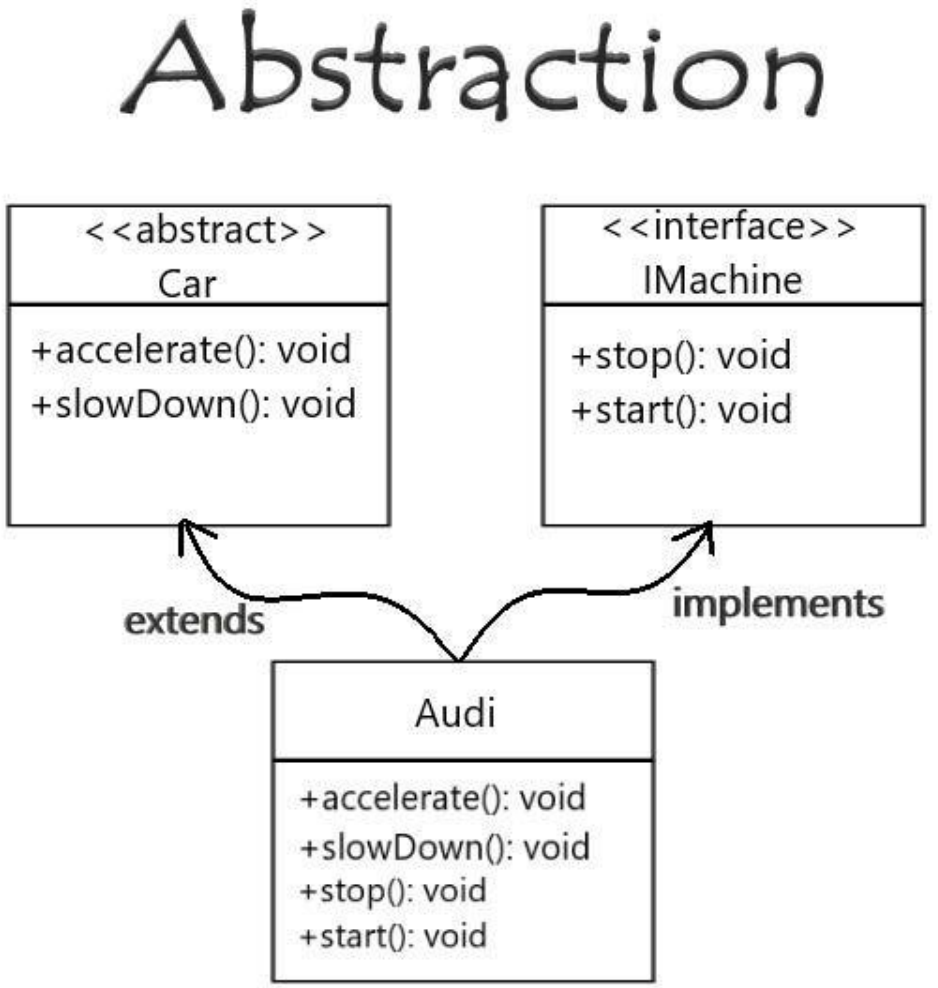
```python
# --- ลองใช้งาน ---
my_car = Car("Toyota")
my_bike = Motorcycle("Honda")

my_car.start_engine()
my_bike.start_engine()
```

```python
from abc import ABC, abstractmethod

# สร้างคลาสแม่แบบ(Abstract Class)
class Vehicle(ABC):
    def __init__(self, brand):
        self.brand = brand

    # ประกาศเมธอดที่บังคับให้คลาสลูกต้องมี
  @abstractmethod
    def start_engine(self):
        pass  # ไม่ต้องเขียนโค้ดข้างใน แค่ประกาศไว้

  @abstractmethod
    def drive(self):
        pass

# --- คลาสลูกที่นำไปใช้งาน ---

class Car(Vehicle):
    # ต้อง implement เมธอดที่ถูกบังคับไว้ให้ครบ
  def start_engine(self):
        print("รถยนต์กำลังสตาร์ทเครื่อง... บรื้นนน!")

  def drive(self):
        print(f"รถยนต์ยี่ห้อ{self.brand} กำลังเคลื่อนที่บนถนน")

class Motorcycle(Vehicle):
    # ต้อง implement เมธอดที่ถูกบังคับไว้ให้ครบ
  def start_engine(self):
        print("มอเตอร์ไซค์กำลังสตาร์ท... แบ๊นๆ!")

  def drive(self):
        print(f"มอเตอร์ไซค์ยี่ห้อ{self.brand} กำลังขับขี่อย่างคล่องแคล่ว"
```

คำถามสำคัญ: ถ้าเราพยายามสร้าง Object จากคลาสแม่แบบโดยตรงแบบนี้ v = Vehicle("some_brand") คิดว่าจะเกิดอะไรขึ้นคะ?

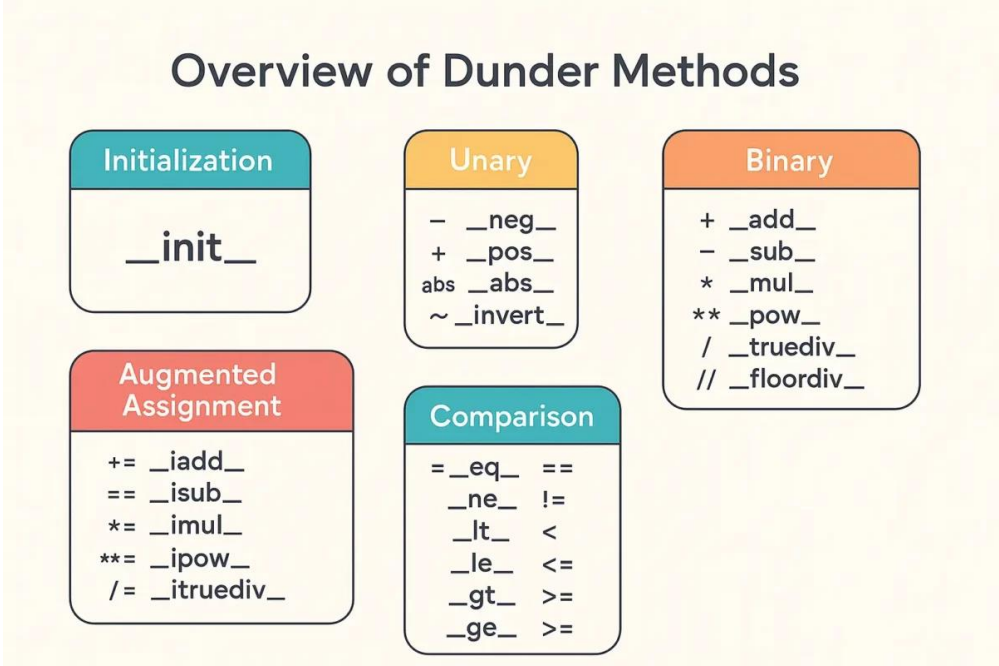
ประโยชน์ของ Abstraction

ประโยชน์ของ Abstraction คือ: ช่วยให้เราออกแบบโครงสร้างของโปรแกรมโดยรวมได้ง่ายขึ้น ทำให้มั่นใจได้ว่า Object ทุกชิ้นที่เกี่ยวข้องจะมีความสามารถพื้นฐานที่จำเป็นครบถ้วน ส่งผลให้โค้ดส่วนอื่นๆ ที่จะเรียกใช้งาน Object เหล่านี้ทำงานร่วมกันได้อย่างราบรื่นค่ะ



Dunder/Magic Methods

- Dunder Methods (มาจาก Double Underscore) คือเมธอดพิเศษที่มีชื่อขึ้นต้นและลงท้ายด้วยขีดล่างสองตัว (__) ครับ เราเรียกมันว่า "Magic Methods" (เมธอดเวทมนตร์) เพราะเรามักจะไม่ได้เรียกใช้มันโดยตรง แต่ Python จะเป็นคนเรียกใช้เมธอดเหล่านี้ให้เราเองโดยอัตโนมัติ เมื่อเราใช้คำสั่งหรือเครื่องหมายบางอย่างกับ Object ของเรามันคือเบื้องหลังที่ทำให้ len([1, 2, 3]) หรือ 'hello' + ' world' ทำงานได้นั่นเองค่ะ



Dunder

ลองจินตนาการว่า Object ของเราเป็นอุปกรณ์ชิ้นหนึ่ง Dunder Methods ก็เปรียบเสมือน **"ปุ่มลับ"** หรือ **"พอร์ตเชื่อมต่อพิเศษ"** ที่ Python รู้จัก

เมื่อคุณพยายามใช้ฟังก์ชัน print() กับ Object ของคุณ Python จะมองหาปุ่มลับที่ชื่อ __str__ แล้วกดให้

เมื่อคุณพยายามใช้เครื่องหมาย + ระหว่าง Object สองชิ้น Python จะมองหาปุ่มลับ __add__ แล้วกดให้

เมื่อคุณพยายามใช้ฟังก์ชัน len() กับ Object ของคุณ Python จะมองหาปุ่มลับ __len__ แล้วกดให้

ถ้า Object ของเรามีปุ่มลับเหล่านี้ มันก็จะสามารถทำงานร่วมกับโลกของ Python ได้อย่างราบรื่น แต่ถ้าไม่มี Python ก็จะบอกว่า "ทำไม่ได้" (raise Error)

ตัวอย่าง: คลาสเวกเตอร์ (Vector)

เรามาสร้างคลาส Vector สำหรับเวกเตอร์ 2 มิติ (ที่มีค่า x และ y) กันค่ะ เราจะเพิ่ม "ปุ่มลับ" เข้าไปเพื่อให้มันบวกกันได้, คูณกับตัวเลขได้ และแสดงผลสวยๆ ได้

v1 คือ Vector(2, 4)
v2 คือ Vector(5, 1)
v1 + v2 = Vector(7, 5)
v1 * 3 = Vector(6, 12)

เห็นมั้ยคะว่าเราไม่ได้เขียนโค้ดเรียก v1.__add__(v2) เลย เราแค่ใช้ v1 + v2 เหมือนกับที่เราบวกตัวเลขธรรมดา แต่เพราะคลาส Vector ของเรามี "ปุ่มลับ" ที่ชื่อ __add__ เตรียมไว้ Python จึงรู้ว่าต้องทำอย่างไรการใช้ Dunder Methods ทำให้ Object ที่เราสร้างขึ้นมามีชีวิตชีวาและทำงานได้เป็นธรรมชาติเหมือนกับ Type ข้อมูลพื้นฐานของ Python เลยค่ะ

```python
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # ปุ่มลับสำหรับ print() และ str()
    # คืนค่าเป็น string ที่คนอ่านเข้าใจง่าย
    def __str__(self):
        return f"Vector({self.x}, {self.y})"

    # ปุ่มลับสำหรับเครื่องหมาย '+'
    def __add__(self, other_vector):
        new_x = self.x + other_vector.x
        new_y = self.y + other_vector.y
        return Vector(new_x, new_y) # คืนค่าเป็น Object Vector ตัวใหม่

    # ปุ่มลับสำหรับเครื่องหมาย '*' (คูณกับตัวเลข)
    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

# --- ลองใช้งาน Magic Methods ---
v1 = Vector(2, 4)
v2 = Vector(5, 1)

# 1. Python จะเรียกใช้ v1.__str__() ให้เอง
print(f"v1 คือ {v1}")
print(f"v2 คือ {v2}")

# 2. Python จะเรียกใช้ v1.__add__(v2) ให้เอง
v3 = v1 + v2
print(f"v1 + v2 = {v3}")

# 3. Python จะเรียกใช้ v1.__mul__(3) ให้เอง
v4 = v1 * 3
print(f"v1 * 3 = {v4}")
```

Class Methods & Static Methods

ปกติแล้ว เมธอดที่เราเขียนในคลาสจะเป็น Instance Methods ซึ่งเป็นเมธอดที่ผูกอยู่กับ Object แต่ละตัว (ที่เราเรียกว่า self) แต่ในบางสถานการณ์ เราอาจต้องการเมธอดที่ทำงานในระดับที่ต่างออกไปครับ Python จึงมีเมธอดอีก 2 ประเภทให้เราใช้

1. Instance Method (เมธอดของอ็อบเจกต์)

นี่คือเมธอดปกติที่เราใช้กันมาตลอดค่ะ มันจะรับ self เป็นพารามิเตอร์ตัวแรกเสมอ และใช้สำหรับทำงานกับข้อมูล (attributes) ของ Object ตัวนั้นๆ

ต้องใช้: self

ผูกกับ: Object (Instance)

ตัวอย่าง: คำนวณพื้นที่ของพิซซ่าถาด *นี้*

2. Class Method (เมธอดของคลาส)

นี่คือเมธอดที่ผูกอยู่กับ **Class** โดยตรง ไม่ใช่ Object แต่ละตัว มันจะรับ cls (ซึ่งหมายถึงตัวคลาสเอง) เป็นพารามิเตอร์ตัวแรกแทน self

ใช้ทำอะไร? ส่วนใหญ่มักใช้เป็น **"Factory Methods"** คือเป็นเมธอดทางเลือกสำหรับสร้าง Object ของคลาสตัวเอง เช่น สร้างพิซซ่าหน้าสำเร็จรูปโดยไม่ต้องระบุส่วนผสมเอง

- **ต้องใช้:** @classmethod decorator และ cls
- **ผูกกับ:** Class
- **ตัวอย่าง:** สร้างพิซซ่าหน้ามาการิต้าตามสูตรมาตรฐานของ *ร้าน*

3. Static Method (เมธอดสถิต)

นี่คือเมธอดที่ **ไม่ผูกกับอะไรเลย** ไม่ว่าจะเป็น Object (self) หรือ Class (cls) มันเป็นเหมือนฟังก์ชันธรรมดาทั่วไปที่บังเอิญมาอยู่ข้างในคลาสเฉยๆ เพื่อความเป็นระเบียบ

ใช้ทำอะไร? ใช้กับฟังก์ชันอรรถประโยชน์ (utility function) ที่มีความเกี่ยวข้องกับคลาส แต่ไม่จำเป็นต้องเข้าถึงข้อมูลของคลาสหรืออ็อบเจกต์เลย

- **ต้องใช้:** @staticmethod decorator
- **ผูกกับ:** ไม่ผูกกับอะไรเลย
- **ตัวอย่าง:** ฟังก์ชันสำหรับตรวจสอบว่าส่วนผสมนี้เป็นมังสวิรัติหรือไม่

ตัวอย่างโค้ด: คลาส Pizza

ผลลัพธ์ที่ได้

Pizza with pepperoni, cheese

Pizza with ham, pineapple, cheese

พิซซ่าหน้าชีสเห็ดเป็นมังสวิรัติหรือไม่? -> True

พิซซ่าของฉันเป็นมังสวิรัติหรือไม่? -> False

สรุปความแตกต่าง

ประเภทเมธอด	ผูกกับ	พารามิเตอร์ตัวแรก	Decorator
Instance Method	Object	self	ไม่มี
Class Method	Class	cls	@classmethod
Static Method	ไม่ผูกเลย	ไม่มี	@staticmethod

```python
class Pizza:
    # 1. Instance Method: ทำงานกับข้อมูลของพิซซ่าแต่ละถาด (self)
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __str__(self):
        return f"Pizza with {', '.join(self.ingredients)}"

    # 2. Class Method: ทำงานกับ Class (cls) เพื่อสร้าง Object
    @classmethod
    def from_hawaiian(cls):
        # cls ในที่นี้ก็คือ class Pizza นั่นเอง
    return cls(['ham', 'pineapple', 'cheese'])

    # 3. Static Method: ฟังก์ชัน is_vegetarian ที่ไม่ต้องการ self หรือ cls
    @staticmethod
    def is_vegetarian(ingredients_list):
        meats = ['ham', 'bacon', 'salami', 'pepperoni']
        for item in ingredients_list:
            if item in meats:
                return False
        return True

# --- ลองใช้งาน ---

# สร้างพิซซ่าแบบปกติ (ใช้ instance method __init__)
my_pizza = Pizza(['pepperoni', 'cheese'])
print(my_pizza)

# สร้างพิซซ่าด้วย Class Method
hawaiian_pizza = Pizza.from_hawaiian()
print(hawaiian_pizza)

# เรียกใช้ Static Method (เรียกผ่าน Class จะเข้าใจง่ายที่สุด)
is_veggie_1 = Pizza.is_vegetarian(['cheese', 'tomato',
'mushroom'])
is_veggie_2 = Pizza.is_vegetarian(my_pizza.ingredients)

print(f"พิซซ่าหน้าชีสเห็ดเป็นมังสวิรัติหรือไม่? -> {is_veggie_1}")
print(f"พิซซ่าของฉันเป็นมังสวิรัติหรือไม่? -> {is_veggie_2}")
```

Composition over Inheritance

- ในโลกของ OOP เราสามารถสร้างความสัมพันธ์ระหว่างคลาสได้ 2 แบบหลักๆ:
- **Inheritance (ความสัมพันธ์แบบ "is-a" / "เป็น"):**
  - คลาสลูก **"เป็น"** คลาสแม่ชนิดหนึ่ง
  - เช่น Car **is a** Vehicle (รถยนต์คือยานพาหนะชนิดหนึ่ง)
  - เป็นความสัมพันธ์ที่ **แน่นแฟ้น (Tightly Coupled)** คลาสลูกจะผูกติดกับโครงสร้างของคลาสแม่ เปลี่ยนแปลงได้ยาก
- **Composition (ความสัมพันธ์แบบ "has-a" / "มี"):**
  - อ็อบเจกต์หนึ่ง **"มี"** อ็อบเจกต์อื่นเป็นส่วนประกอบ
  - เช่น Car **has an** Engine (รถยนต์มีเครื่องยนต์)
  - เป็นความสัมพันธ์ที่ **ยืดหยุ่น (Loosely Coupled)** เราสามารถสับเปลี่ยนชิ้นส่วนได้ง่ายกว่า
- **ปรัชญานี้บอกว่า:** *เมื่อเราต้องการสร้างคลาสใหม่ที่มีความสามารถคล้ายของเดิม ให้ลองคิดก่อนว่าเราจะสร้างมันขึ้นมาโดยการ "นำชิ้นส่วนมาประกอบกัน" (Composition) ได้หรือไม่ ก่อนที่จะคิดว่ามัน "เป็น" อีกคลาสหนึ่ง (Inheritance)*

ทำไมถึงเป็นเช่นนั้น?

- การใช้ Inheritance พร่ำเพรื่ออาจนำไปสู่ปัญหาได้ค่ะ:
- **ความซับซ้อน:** อาจเกิดสายการสืบทอดที่ยาวเหยียดและซับซ้อน (คลาส A สืบทอดไป B ไป C ไป D...) ทำให้ทำความเข้าใจและแก้ไขได้ยาก
- **ความตายตัว:** คลาสลูกจะถูกผูกมัดกับคลาสแม่ทั้งหมด ถ้าคลาสแม่เปลี่ยนแปลง อาจส่งผลกระทบพังลงมาเป็นทอดๆ ได้
- **ปัญหา "กล้วยกับกอริลลา":** "คุณแค่อยากได้กล้วย แต่กลับได้กอริลลาที่ถือกล้วยและป่าทั้งป่ามาด้วย" บางครั้งเราแค่อยากได้ความสามารถแค่ 1-2 อย่างจากคลาสแม่ แต่ Inheritance บังคับให้เราต้องรับมาทั้งหมด

ตัวอย่างโค้ด: การสร้างหุ่นยนต์

สมมติเราต้องการสร้างหุ่นยนต์ที่ "เคลื่อนที่ได้" และ "พูดได้"วิธีที่ 1: ใช้ Composition (วิธีที่แนะนำ)เราจะสร้างความสามารถแต่ละอย่างเป็นคลาส "ชิ้นส่วน" แยกกัน แล้วนำมาประกอบร่างในคลาส Robot

ข้อดีของวิธีนี้คือ: ถ้าในอนาคตเราอยากสร้างหุ่นยนต์ที่ **บินได้** แทนที่จะเดิน เราก็แค่สร้างคลาส FlyingModule ขึ้นมาใหม่ แล้วตอนสร้าง Robot ก็แค่เปลี่ยนชิ้นส่วน self.movement ให้เป็น FlyingModule() โดยที่ไม่ต้องไปยุ่งกับคลาส Robot หลักเลย! มันยืดหยุ่นมากๆ

วิธีที่ 2: ใช้ Inheritance (อาจเกิดปัญหา) เราอาจสร้างคลาส CanMove และ CanSpeak แล้วให้ Robot สืบทอดจากทั้งสองคลาส ซึ่งเรียกว่า Multiple Inheritance และมักจะนำมาซึ่งความซับซ้อนและปัญหาได้ง่ายกว่ามาก

```
# --- สร้างคลาส "ชิ้นส่วน" ความสามารถ ---
class MovementModule:
    def move(self):
        print("กำลังเคลื่อนที่ไปข้างหน้า...")

class SpeakerModule:
    def speak(self, message):
        print(f"Robot says: {message}")


# --- นำชิ้นส่วนมา "ประกอบร่าง" เป็นหุ่นยนต์ ---
class Robot:
    def __init__(self):
        # Robot "มี" MovementModule และ
SpeakerModule เป็นส่วนประกอบ
        self.movement = MovementModule()
        self.speaker = SpeakerModule()

    # มอบหมายงานให้ชิ้นส่วนที่รับผิดชอบโดยตรง
  def move(self):
        self.movement.move()

    def speak(self, message):
        self.speaker.speak(message)


# --- ลองใช้งาน ---
my_robot = Robot()
my_robot.move()
my_robot.speak("Hello World!")
```

Inheritance ไม่ใช่สิ่งเลวร้าย! มันยังมีประโยชน์มากสำหรับความสัมพันธ์แบบ "is-a" ที่ชัดเจนจริงๆ (เช่น Cat is an Animal) แต่ปรัชญา "Composition over Inheritance" เป็นเครื่องเตือนใจให้เราออกแบบคลาสโดยนึกถึงความยืดหยุ่นเป็นอันดับแรกเสมอ