

OOP;



OOP

```
unittest.mock;\nfrom unittest\nimport *\n\nif __name__ == '__main__':\n    pass
```

```
function)\n//ist;\nprogramming)\n}
```

AI

Generative AI

TAE

Fioetice

PYTHON PROGRAMMING: ADVANCE (INTEGRATIVE-GENERATIVE AI EDITION)



Asynhroous
programming



Unit Testing

Student Price Book Center

คำนำ

ในยุคที่เทคโนโลยีซอฟต์แวร์เติบโตอย่างรวดเร็ว การเขียนโปรแกรมไม่ใช่เพียงแค่การสั่งให้คอมพิวเตอร์ทำงานตามลำดับเท่านั้น หากแต่ยังต้องคำนึงถึงความยืดหยุ่น ความถูกต้อง ความสามารถในการปรับขยาย และความเสถียรของระบบในระยะยาว หนังสือวิชาการเล่มนี้จึงได้รับการเรียบเรียงขึ้นเพื่อปูพื้นฐานแนวคิดและเทคนิคขั้นสูงในการเขียนโปรแกรมด้วยภาษา Python โดยเน้นแนวทางการพัฒนาเชิงโครงสร้างที่มีระบบแบบมืออาชีพในระดับ **Advanced Level** ที่ครอบคลุมทั้งด้านการออกแบบ การประมวลผล และการตรวจสอบคุณภาพของโค้ด

เริ่มต้นจากหัวข้อสำคัญอย่าง **การเขียนโปรแกรมเชิงวัตถุ (OOP)** ที่เป็นพื้นฐานในการออกแบบระบบซอฟต์แวร์ที่มีโครงสร้างชัดเจนและยืดหยุ่น ผู้เรียนจะได้เข้าใจหลักการสร้างคลาส การใช้ `__init__`, `self`, และ attribute ต่าง ๆ พร้อมทั้งเรียนรู้การสืบทอด (inheritance), การห่อหุ้ม (encapsulation), และความหลากหลายของพฤติกรรม (polymorphism) ตลอดจนการใช้ magic methods เพื่อควบคุมพฤติกรรมของอ็อบเจกต์ให้เหมาะสมกับการใช้งานจริง

ต่อมาเป็นการเปิดโลกสู่แนวคิดแบบ **Functional Programming (FP)** ที่เน้นฟังก์ชันบริสุทธิ์ การไม่เปลี่ยนแปลงสถานะภายนอก (immutability) และการจัดการข้อมูลอย่างปลอดภัยผ่านฟังก์ชัน `map()`, `filter()`, และ `reduce()` รวมถึงการใช้ `functools` module เพื่อเพิ่มประสิทธิภาพในการคำนวณ นอกจากนี้ยังครอบคลุมแนวคิดเชิงลึก เช่น monad-like patterns และ function composition ซึ่งช่วยให้โค้ดอ่านง่าย ปลอดภัย และมีความสามารถในการประมวลผลอย่างเป็นระบบ

ในส่วนของการเขียนโปรแกรมแบบ **asynchronous** ผู้อ่านจะได้เรียนรู้การใช้งาน `async/await`, การจัดการ event loop ด้วย `asyncio`, และการประยุกต์ใช้กับ I/O แบบไม่ประสานเวลา เพื่อเพิ่มประสิทธิภาพการทำงานและรองรับการประมวลผลพร้อมกันจำนวนมากในระบบที่ต้องการความเร็วและตอบสนองแบบเรียลไทม์

ท้ายที่สุด หนังสือเล่มนี้ยังครอบคลุมหัวข้อ **การทดสอบและตรวจสอบโค้ด** เพื่อรับประกันว่าโค้ดที่เขียนมีความถูกต้องและพร้อมสำหรับการใช้งานจริง โดยแนะนำการใช้ `unittest` และ `pytest` สำหรับ unit testing, การ mock ข้อมูลเพื่อทดสอบแบบแยกส่วน การวิเคราะห์ test coverage ตลอดจนการใช้ type hinting และเครื่องมือตรวจสอบ static อย่าง `mypy` เพื่อยกระดับคุณภาพและความเข้าใจของโค้ดให้ดียิ่งขึ้น

ผู้เขียนเชื่อว่าผู้อ่านที่มีพื้นฐาน Python และต้องการยกระดับทักษะสู่ระดับสูงจะได้รับประโยชน์อย่างแท้จริงจากหนังสือเล่มนี้ ไม่เพียงแต่สามารถพัฒนาโปรแกรมที่ซับซ้อนและมีประสิทธิภาพได้ แต่ยังสามารถสร้างโค้ดที่ดูแลรักษาได้ดี มีมาตรฐาน และพร้อมสำหรับการประยุกต์ใช้ในโลกของการพัฒนาซอฟต์แวร์ระดับมืออาชีพ.

ด้วยความปรารถนาดี
ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 11 การเขียนโปรแกรมเชิงวัตถุ (Object Oriented Programming: OOP).....	1
---	---

- พื้นฐานการสร้าง class และ object
- แนวคิดเชิงลึก: OOP ใน Python
- การเปรียบเทียบ UML (Unified Modeling Language) กับ OOP ในภาษา Python
- การสร้าง class และ object ในภาษา Python อย่างละเอียด
- การสร้าง Class และ Object เชิงลึก + UML
- การใช้ __init__, self และ attribute ต่าง ๆ ใน Python OOP — อธิบายอย่างละเอียด
- การสืบทอดคลาส (Inheritance) ใน Python
- Polymorphism และ Encapsulation ในเชิงลึก พร้อมตัวอย่าง และ UML
- อธิบายเชิงลึกเกี่ยวกับ Magic Methods ใน Python

บทที่ 12 การเขียนโปรแกรมเชิงฟังก์ชัน (Functional Programming)	84
---	----

- พื้นฐานการเขียนโปรแกรมเชิงฟังก์ชัน (Functional Programming)
- เจาะลึกเพิ่มเติม การเขียนโปรแกรมเชิงฟังก์ชัน (Functional Programming) ใน Python
- Pure Functions (ฟังก์ชันบริสุทธิ์)
- เนื้อหาเชิงลึกและตัวอย่างในหัวข้อ Pure Functions และ Immutability พร้อมแนะนำแนวทางประยุกต์ใช้งานจริง
- ตัวอย่างโปรแกรมแนวประยุกต์ใช้งานจริง
- อธิบายเนื้อหา map, filter, และ reduce ให้ละเอียดเชิงลึก
- อธิบาย functools module, lru_cache, partial, และ wraps
- อธิบาย functools.lru_cache, functools.partial, functools.wraps เชิงลึก
- Monad-like patterns และ Function Composition ในภาษา Python
- ตัวอย่างโปรแกรมบูรณาการ

บทที่ 13 การเขียนโปรแกรมแบบ asynchronous (Asynchronous Programming).....	145
--	-----

- พื้นฐานการเขียนโปรแกรมแบบ Asynchronous (Asynchronous Programming)
- การเขียนโปรแกรมแบบ Asynchronous ใน Python — เชิงลึก

- อธิบายเนื้อหา async / await ใน Python อย่างละเอียด
- เพิ่มเนื้อหาเชิงลึกและตัวอย่างการใช้งาน async / await ใน Python ที่หลากหลาย
- asyncio กับ event loop
- เสริมเนื้อหาเชิงลึกและตัวอย่างเกี่ยวกับ asyncio กับ event loop
- การประยุกต์ใช้งานจริงของ asyncio + async/await + event loop
- การใช้งาน async ร่วมกับ IO (Input/Output) ใน Python
- เพิ่มเติม: การใช้งาน async กับ IO ในสถานการณ์จริงและตัวอย่างเชิงลึก

บทที่ 14 การทดสอบและตรวจสอบโค้ด (Unit testing and Test Coverage) 208

- พื้นฐานการทดสอบและตรวจสอบโค้ด
- การทดสอบและตรวจสอบโค้ด (Testing and Code Verification) เพิ่มเติม
- เนื้อหาเกี่ยวกับ Unit testing ด้วย unittest และ pytest แบบละเอียด
- Mocking
- การทำ Test Coverage
- Type Hint ใน Python และการใช้เครื่องมือ Static Type Checker อย่าง mypy
- Static Type Checker: mypy
- ตัวอย่างโปรแกรมบูรณาการ

บรรณานุกรม 260

บทที่ 11

การเขียนโปรแกรมเชิงวัตถุ (Object Oriented Programming: OOP)

เนื้อหา

- พื้นฐานการสร้าง class และ object
- แนวคิดเชิงลึก: OOP ใน Python
- การเปรียบเทียบ UML (Unified Modeling Language) กับ OOP ในภาษา Python
- การสร้าง class และ object ในภาษา Python อย่างละเอียด
- การสร้าง Class และ Object เชิงลึก + UML
- การใช้ `__init__`, `self` และ attribute ต่าง ๆ ใน Python OOP — อธิบายอย่างละเอียด
- การสืบทอดคลาส (Inheritance) ใน Python
- Polymorphism และ Encapsulation ในเชิงลึก พร้อมตัวอย่าง และ UML
- อธิบายเชิงลึกเกี่ยวกับ Magic Methods ใน Python

บทนำ

ในการเขียนโปรแกรมยุคใหม่ การพัฒนาโปรแกรมที่สามารถจัดการกับข้อมูลและพฤติกรรมได้อย่างเป็นระบบถือเป็นสิ่งสำคัญอย่างยิ่ง ซึ่งแนวคิด “การเขียนโปรแกรมเชิงวัตถุ” หรือ OOP (Object-Oriented Programming) คือแนวทางหลักที่ช่วยให้นักพัฒนาสามารถออกแบบซอฟต์แวร์ที่มีความยืดหยุ่น ขยายง่าย และบำรุงรักษาได้ดี ในบทนี้ ผู้อ่านจะได้เรียนรู้แนวคิดเชิงวัตถุที่เป็นหัวใจสำคัญของการพัฒนาโปรแกรม Python สมัยใหม่ และสามารถนำไปประยุกต์ใช้ได้อย่างมั่นใจในงานจริง

เริ่มต้นจากพื้นฐานของการสร้าง **คลาส (class)** และ **วัตถุ (object)** ซึ่งเป็นองค์ประกอบหลักของ OOP โดยผู้อ่านจะได้เข้าใจการนิยามคลาส การใช้เมธอด `__init__`, ตัวแปร `self`, และการกำหนดแอตทริบิวต์ให้กับวัตถุ ซึ่งเป็นก้าวแรกในการแปลงแนวคิดเชิงปัญหาให้อยู่ในรูปแบบของโครงสร้างโปรแกรมที่จับต้องได้และมีความหมาย

จากนั้น บทเรียนจะนำเสนอหลักการสำคัญของ OOP ได้แก่ **การสืบทอดคลาส (Inheritance)** ซึ่งช่วยให้สามารถสร้างคลาสใหม่ที่สืบทอดพฤติกรรมจากคลาสเดิมได้โดยไม่ต้องเขียนโค้ดซ้ำ และ **Polymorphism** ซึ่งเปิดโอกาสให้คลาสต่าง ๆ ใช้อินเทอร์เฟซเดียวกันในการทำงาน ร่วมกับ **Encapsulation** ที่ช่วยซ่อนรายละเอียดการทำงานภายในวัตถุเพื่อให้โค้ดมีความปลอดภัยและง่ายต่อการควบคุม

อีกหนึ่งจุดเด่นของ Python คือการรองรับ **Magic Methods** หรือ dunder methods (เช่น `__str__`, `__repr__`, `__len__`) ที่ช่วยให้สามารถควบคุมพฤติกรรมของวัตถุเมื่อถูกใช้ร่วมกับฟังก์ชันภายนอกหรือโอเปอเรเตอร์ต่าง ๆ ได้อย่างยืดหยุ่น โดยการทำความเข้าใจกลไกเหล่านี้จะช่วยให้ผู้อ่านสามารถปรับแต่งพฤติกรรมของคลาสได้อย่างลึกซึ้งและเป็นธรรมชาติยิ่งขึ้น

บทนี้จึงเหมาะอย่างยิ่งสำหรับผู้ที่ต้องการยกระดับจากการเขียนโปรแกรมเชิงโครงสร้างไปสู่การเขียนโปรแกรมเชิงวัตถุอย่างเต็มรูปแบบ เพื่อเตรียมพร้อมสำหรับการสร้างระบบซอฟต์แวร์ที่มีความซับซ้อน มีการจัดการข้อมูลอย่างเป็นระบบ และสามารถขยายต่อขยายได้อย่างยั่งยืน ผู้เขียนหวังว่าความรู้ที่ได้นำเสนอในบทนี้จะเป็นรากฐานสำคัญในการสร้างความเข้าใจที่ถูกต้อง และเป็นแรงบันดาลใจให้ผู้อ่านพัฒนาไปสู่การเป็นนักพัฒนาซอฟต์แวร์มืออาชีพในอนาคต.

พื้นฐานการสร้าง class และ object

➤ ความหมาย

- class คือแม่แบบ (template) สำหรับสร้างวัตถุ (object)
- object คืออินสแตนซ์ (instance) ของ class ซึ่งสามารถมี **attribute** (ข้อมูล) และ **method** (ฟังก์ชัน)

➤ ตัวอย่าง

```
class Person:
```

```
    def speak(self):
        print("Hello!")
```

```
# สร้าง object
```

```
p1 = Person()
```

```
p1.speak()
```

❑ ผลลัพธ์:

```
Hello!
```

❑ 2. การใช้ `__init__`, `self` และ attribute ต่าง ๆ

➤ คำอธิบาย:

- `__init__` คือ constructor ทำงานเมื่อสร้าง object
- `self` คือการอ้างถึงตัว object เอง
- attribute คือข้อมูลประจำ object (เช่น name, age)

➤ ตัวอย่าง:

```
class Person:
```

```
def __init__(self, name, age):
    self.name = name
    self.age = age

def info(self):
    print(f"ชื่อ: {self.name}, อายุ: {self.age}")
```

```
p2 = Person("Alice", 25)
```

```
p2.info()
```

☐ ผลลัพธ์:

ชื่อ: Alice, อายุ: 25

☐ 3. การสืบทอดคลาส (Inheritance)

➤ ความหมาย:

คลาสลูก (subclass) สามารถ สืบทอด คุณสมบัติของคลาสแม่ (superclass) ได้ และสามารถเพิ่มหรือ override method ได้

➤ ตัวอย่าง:

```
class Animal:
```

```
    def speak(self):
        print("เสียงสัตว์")
```

```
class Dog(Animal):
```

```
    def speak(self):
        print("โห่ง!")
```

```
a = Animal()
```

```
d = Dog()
```

```
a.speak() # จากคลาสแม่
```

```
d.speak() # override
```

☐ ผลลัพธ์:

เสียงสัตว์

โห่ง!

☐ 4. Polymorphism และ Encapsulation

☐ Polymorphism (พหุรูปแบบ):

- ความสามารถของ object ที่มี method ชื่อเดียวกัน แต่พฤติกรรมต่างกันตามคลาส

class Cat:

```
def speak(self):
    print("เหมียว")
```

class Dog:

```
def speak(self):
    print("โห้ง")
```

```
def animal_speak(animal):
```

```
    animal.speak()
```

```
animal_speak(Cat())
```

```
animal_speak(Dog())
```

☐ ผลลัพธ์:

เหมียว

โห้ง

☐ Encapsulation (การซ่อนข้อมูล):

- ใช้ `_` หรือ `__` เพื่อกำหนดระดับการเข้าถึง (protected, private)

class BankAccount:

```
def __init__(self, owner, balance):
```

```
    self.owner = owner
```

```
    self.__balance = balance # private
```

```
def deposit(self, amount):
```

```
    self.__balance += amount
```

```
def get_balance(self):
```

```
    return self.__balance
```

```
acct = BankAccount("Bob", 1000)
```

```
acct.deposit(500)
```

```
print(acct.get_balance())
```

```
# print(acct.__balance) ☐ Error
```

☐ ผลลัพธ์:

```
1500
```

☐ 5. Magic Methods (__str__, __repr__, __len__, etc.)

➤ ความหมาย:

เมธอดพิเศษที่ Python ใช้ในการควบคุมการแสดงผล การเปรียบเทียบ การทำงานร่วมกับฟังก์ชัน built-in เช่น len(), print()

➤ ตัวอย่าง:

```
class Book:
```

```
    def __init__(self, title, pages):
```

```
        self.title = title
```

```
        self.pages = pages
```

```
    def __str__(self):
```

```
        return f"[Book] {self.title}"
```

```
    def __len__(self):
```

```
        return self.pages
```

```
b = Book("Python OOP", 300)
```

```
print(b)
```

```
print(len(b))
```

☐ ผลลัพธ์:

```
[Book] Python OOP
```

```
300
```

☐ สรุปความเข้าใจ (Concept Map):

แนวคิด OOP	คำอธิบาย
Class/Object	แบบพิมพ์ของข้อมูลและอินสแตนซ์จริง
__init__	สร้าง object พร้อมค่าตั้งต้น
self	อ้างอิง object ปัจจุบัน

แนวคิด OOP	คำอธิบาย
Attribute	ตัวแปรภายใน object
Inheritance	การถ่ายทอดคุณสมบัติ
Polymorphism	การใช้ method ชื่อเดียวในหลายรูปแบบ
Encapsulation	การควบคุมการเข้าถึงข้อมูล
Magic Methods	การควบคุมพฤติกรรมพิเศษของ object

แนวคิดเชิงลึก: OOP ใน Python

- ☐ 1. class และ object ในระดับลึก
- ☐ OOP ใน Python คือการจำลองโลกแห่งความเป็นจริง ผ่านการออกแบบวัตถุที่มี สถานะ (State) และ พฤติกรรม (Behavior)
 - Python ไม่มี "access modifier" แบบ public/private จริง แต่ใช้ชื่อตกลง (`_name`, `__name`) แทน
 - คลาสใน Python เป็น **first-class object**: เราสามารถส่งผ่าน ฟังก์ชัน สร้าง runtime หรือแม้แต่ซ่อน class ได้
- ☐ เทคนิคขั้นสูง:
 - ใช้ **metaclass** เพื่อสร้าง class แบบไดนามิก
 - คลาสสามารถเก็บ static attribute, dynamic attribute, หรือใช้ descriptor เพื่อควบคุม property ได้

☐ 2. `__init__`, `self`, และ attribute

- ☐ ในระดับลึก:
 - `__init__()` เป็น initializer ไม่ใช่ constructor จริงๆ — constructor ที่แท้คือ `__new__()`
 - `self` คือ object instance ที่ถูกสร้างขึ้น และจะถูกผูก binding กับ attribute และ method ภายใน object
 - สามารถใช้ `__slots__` เพื่อจำกัด attribute ที่ object จะมี ช่วยประหยัด memory

class Slim:

```
__slots__ = ['x', 'y'] # ป้องกันการเพิ่ม attribute อื่นๆ
def __init__(self, x, y):
    self.x = x
    self.y = y
```

☐ 3. การสืบทอด (Inheritance) เชิงลึก

☐ Multiple Inheritance:

- Python รองรับ multiple inheritance โดยใช้ **C3 Linearization (MRO)**

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
```

```
print(D.__mro__)
```

☐ ผลลัพธ์:

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
<class 'object'>)
```

☐ เทคนิคเชิงลึก:

- ใช้ `super()` แบบ dynamic เพื่อแก้ปัญหา diamond problem
- ใช้ `@abstractmethod` ใน base class เพื่อ enforce contract

☐ 4. Polymorphism และ Encapsulation

☐ Polymorphism:

- ฟังก์ชันเดียวกันทำงานต่างกันตาม object ที่ใช้
- สามารถใช้ `@staticmethod`, `@classmethod`, และ `functools.singledispatch` เพื่อเพิ่มความยืดหยุ่น

```
from functools import singledispatch
```

```
@singledispatch
```

```
def operate(data):
```

```
    raise NotImplementedError
```

```
@operate.register
```

```
def _(data: int):
```

```
    return data * 2
```

```
@operate.register
```

```
def _(data: str):
```

```
return data.upper()
```

☐ ผล:

```
operate(10) # 20
```

```
operate("hi") # HI
```

☐ **Encapsulation:**

- Python ไม่มี keyword private — ใช้ `_` (protected), `__` (name mangling) แทน
- ใช้ `@property` และ `setter` เพื่อควบคุมการเข้าถึงแบบอ่าน-เขียน

```
class Account:
```

```
    def __init__(self, balance):
        self.__balance = balance
```

```
    @property
```

```
    def balance(self):
        return self.__balance
```

```
    @balance.setter
```

```
    def balance(self, amount):
        if amount >= 0:
            self.__balance = amount
```

☐ **5. Magic Methods** เชิงลึก

☐ **Magic Methods หรือ Dunder Methods (Double Underscore)**

- ควบคุมพฤติกรรมของ object เมื่อใช้กับ built-in เช่น `+`, `==`, `len()`, `print()`

```
class Vector:
```

```
    def __init__(self, x, y): self.x, self.y = x, y
    def __add__(self, other): return Vector(self.x + other.x, self.y + other.y)
    def __str__(self): return f'({self.x}, {self.y})'
```

```
v1 = Vector(1, 2)
```

```
v2 = Vector(3, 4)
```

```
print(v1 + v2) # (4, 6)
```

☐ **Magic Methods ที่ควรรู้:**

Method	ใช้สำหรับ
<code>__init__</code>	Initializer (constructor)
<code>__new__</code>	สร้าง object (ก่อน init)

Method	ใช้สำหรับ
<code>__str__</code>	แสดงผลเมื่อใช้ <code>print()</code>
<code>__repr__</code>	แสดงผลใน interactive console
<code>__len__</code>	<code>len(obj)</code>
<code>__getitem__</code>	<code>obj[index]</code>
<code>__setitem__</code>	<code>obj[index] = value</code>
<code>__call__</code>	ทำให้ object เรียกเหมือนฟังก์ชัน
<code>__eq__</code> , <code>__lt__</code> , <code>__hash__</code>	ใช้ใน comparison และ set/dict
<code>__enter__</code> , <code>__exit__</code>	ใช้กับ with (context manager)

☐ การประยุกต์ใช้ OOP ในงานจริง

สถานการณ์จริง	OOP ที่ใช้
ระบบจัดการร้านค้า	Product, Customer, Order class
เกม	Character, Enemy, Weapon, Scene class
เว็บแอป	MVC: Model, View, Controller เป็น class
เครื่องมือประมวลผลข้อมูล	สร้าง Pipeline และ Transformer class
AI/ML pipeline	Model, Dataset, Trainer class

☐ สรุป

การใช้ OOP ใน Python ไม่ใช่แค่การ “รวมโค้ดในคลาส” แต่เป็นการออกแบบซอฟต์แวร์ให้มี **ความยืดหยุ่น**, การขยายตัวได้ (**extensible**) และ การดูแลรักษา (**maintainable**) โดยใช้เทคนิคขั้นสูง เช่น:

- Multiple inheritance กับ MRO
- Abstract Base Classes (ABC)
- Encapsulation ด้วย property
- Polymorphism ด้วย singledispatch
- Magic methods เพื่อ custom behavior

การเปรียบเทียบ UML (Unified Modeling Language) กับ OOP ในภาษา Python

☐ UML คืออะไร?

UML คือภาษาสัญลักษณ์มาตรฐานสำหรับการ "ออกแบบและอธิบาย" โครงสร้างของระบบซอฟต์แวร์

- ใช้สำหรับวาด Diagram เพื่อสื่อสารโครงสร้าง คลาส ความสัมพันธ์ และพฤติกรรมของวัตถุ
- เป็นภาษากลางระหว่าง "นักวิเคราะห์" และ "นักพัฒนา"

☐ OOP ใน Python คืออะไร?

OOP (Object-Oriented Programming) คือแนวทางการเขียนโปรแกรมโดยใช้ "คลาส" และ "วัตถุ" ที่จำลองพฤติกรรมในโลกจริง

Python สนับสนุน OOP แบบเต็มรูปแบบ แต่เน้นความยืดหยุ่น ไม่บังคับ strict typing และ visibility (เช่น public/private)

☐ การจับคู่ UML กับ OOP Python

UML Concept	OOP ใน Python	ตัวอย่างใน Python
Class Diagram	class definition	class Animal: ...
Attributes	instance variables	self.name = name
Operations	methods (function in class)	def bark(self):
Generalization	Inheritance	class Dog(Animal):
Association	instance references another class	self.owner = Person()
Aggregation/Composition	has-a relationship	self.engine = Engine()
Multiplicity	list of objects	self.books = []
Visibility	public/protected/private (Python ไม่มี แต่ใช้ convention _, __)	self._id, self.__secret
Abstract Class	ABC module	class Shape(ABC):
Interface	abstract base class with abstract methods	@abstractmethod

☐ ตัวอย่างจับคู่ UML กับ Python

☐ UML Class Diagram:

+-----+

| Animal |

```

+-----+
| - name: str    |
| - age: int     |
+-----+
| +speak(): str  |
+-----+

```



```

+-----+
|   Dog   |
+-----+
| - breed: str |
+-----+
| +speak(): str |
+-----+

```

☐ Python Code:

```
class Animal:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def speak(self):
```

```
        return "Sound"
```

```
class Dog(Animal):
```

```
    def __init__(self, name, age, breed):
```

```
        super().__init__(name, age)
```

```
        self.breed = breed
```

```
    def speak(self):
```

```
        return "Woof"
```

☐ แนวทางบูรณาการ: UML → OOP Python

1. วิเคราะห์ระบบ ด้วย Use Case Diagram

2. ออกแบบ **Class Diagram** เพื่อเห็นโครงสร้างคลาสและความสัมพันธ์
3. ถอดเป็นโค้ด **Python OOP** โดยแมปคลาสและเมธอดตาม UML
4. ใช้ **Sequence Diagram** เพื่อวางลำดับการเรียกเมธอด
5. ทดสอบและขยายคลาส ด้วย Inheritance และ Polymorphism

☐ สรุปเปรียบเทียบ

ด้าน	UML	OOP Python
จุดประสงค์	ออกแบบและสื่อสารโครงสร้าง	ลงมือเขียนโปรแกรมจริง
การมองภาพ	นามธรรม (abstract)	รูปธรรม (concrete)
ความสำคัญ	ใช้ในการวางแผนล่วงหน้า	ใช้ในการ implement ระบบ
การใช้งาน	เหมาะกับระบบขนาดใหญ่, มีหลายทีม	เขียนเดี่ยว หรือปรับตาม UML

การสร้าง class และ object ในภาษา Python อย่างละเอียด

☐ ความหมายเบื้องต้น

- **Class (คลาส):** แม่แบบสำหรับสร้างวัตถุ (object) ซึ่งกำหนด **attribute** (คุณสมบัติ) และ **method** (พฤติกรรม)
- **Object (วัตถุ):** สิ่งที่สร้างมาจากคลาส เป็นอินสแตนซ์ของคลาส

Python เป็นภาษา **object-oriented** ที่ใช้ class เพื่อจัดโครงสร้างโค้ดให้เป็นระบบและสามารถนำกลับมาใช้ใหม่ได้ง่าย

☐ โครงสร้างพื้นฐานของคลาส

class ClassName:

```
def __init__(self, parameters):
    self.attribute = value
```

```
def method_name(self):
```

```
# พฤติกรรมของวัตถุ
```

- `__init__()` คือ **Constructor** ที่ถูกเรียกอัตโนมัติเมื่อสร้าง object
- `self` คือการอ้างถึงตัว object เอง (จำเป็นต้องมีในเมธอดทุกตัว)

☐ ตัวอย่างที่ 1: การสร้างคลาสและสร้างอ็อบเจกต์

☐ โค้ด:

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

    def introduce(self):
        return f"My name is {self.name}, I am {self.age} years old."

# สร้างอ็อบเจกต์
p1 = Person("Alice", 25)
p2 = Person("Bob", 30)

# เรียกใช้ method
print(p1.introduce())
print(p2.introduce())
```

☐ ผลรัน:

My name is Alice, I am 25 years old.
 My name is Bob, I am 30 years old.

☐ คำอธิบายโค้ด:

- class Person: คือการประกาศคลาสชื่อ Person
- __init__() กำหนดว่าต้องรับ name และ age และกำหนดเป็น attribute ของ object
- self.name = name หมายถึง "เก็บค่าชื่อไว้ใน object นี้"
- introduce() เป็น method ของคลาส ใช้ self เพื่อเข้าถึงข้อมูลใน object
- p1 = Person("Alice", 25) คือการสร้าง object จากคลาส Person
- p1.introduce() เรียกใช้ method

☐ การสร้างหลายอ็อบเจกต์

คุณสามารถสร้าง object ได้หลายตัวจากคลาสเดียวกัน และแต่ละตัวจะมีค่าภายในเป็นของตนเอง

```
students = [
    Person("John", 18),
    Person("Jane", 20),
    Person("Jim", 19)
```

]

for s in students:

```
    print(s.introduce())
```

☐ แนวคิด OOP ที่เริ่มต้นจาก class/object

แนวคิด	อธิบาย
Encapsulation	ซ่อนข้อมูลไว้ในคลาสด้วย self.attribute
Reusability	สร้าง object ที่ตัวก็ได้จากคลาสเดียว
Structure	ได้ดูมีโครงสร้าง แยกพฤติกรรมจากข้อมูลได้

☐ ตัวอย่างประยุกต์ใช้งานจริง

เช่น สร้างระบบนักเรียน, สินค้า, พนักงาน, ยานพาหนะ ฯลฯ โดยใช้คลาสเป็นแม่แบบ

class Product:

```
    def __init__(self, name, price):
```

```
        self.name = name
```

```
        self.price = price
```

```
    def discount(self, percent):
```

```
        return self.price * (1 - percent/100)
```

```
item = Product("Mouse", 500)
```

```
print(f"ลด 10% เหลือ {item.discount(10)} บาท")
```

☐ ผลรัน:

ลด 10% เหลือ 450.0 บาท

☐ ข้อแนะนำสำหรับผู้เริ่มต้น

- ใช้ชื่อคลาสเป็น **PascalCase** เช่น Person, StudentRecord
- ใช้ self เสมอใน method ภายในคลาส
- เรียกใช้ method ด้วย object.method()

การสร้าง Class และ Object เชิงลึก + UML

☐ แนวคิดเบื้องหลัง OOP

Class คือ:

แม่แบบสำหรับการสร้างวัตถุ (Object) ที่กำหนด Attribute และ Method ที่วัตถุนั้นจะมี

Object คือ:

อินสแตนซ์ของคลาสที่มีข้อมูลและพฤติกรรมตามที่กำหนดไว้ในคลาส

☐ ความสัมพันธ์ UML พื้นฐานของ Class

UML Components

- **Class Name** อยู่บนสุด
- **Attributes** อยู่ตรงกลาง
- **Methods** อยู่ล่างสุด
- ใช้ + (public), - (private), # (protected)

☐ ตัวอย่าง Class: Person และ Student

class Person:

```
def __init__(self, name: str, age: int):
    self.name = name
    self.age = age

def introduce(self) -> str:
    return f"My name is {self.name}, I'm {self.age} years old."
```

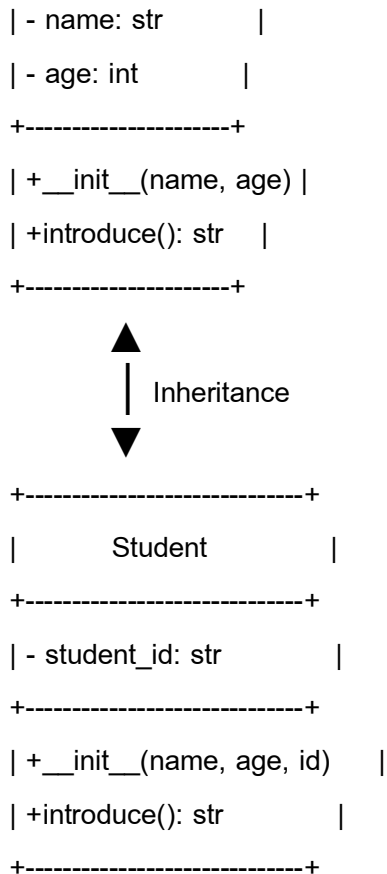
class Student(Person):

```
def __init__(self, name: str, age: int, student_id: str):
    super().__init__(name, age)
    self.student_id = student_id

def introduce(self) -> str:
    return f"I'm {self.name}, {self.age} years old, ID: {self.student_id}"
```

☐ UML Diagram

```
+-----+
|   Person   |
+-----+
```



☐ การใช้งาน:

```

person = Person("Alice", 40)
student = Student("Bob", 20, "S12345")

```

```

print(person.introduce())
print(student.introduce())

```

☐ ผลลัพธ์:

```

My name is Alice, I'm 40 years old.
I'm Bob, 20 years old, ID: S12345

```

☐ อธิบายเชิงลึก

☐ การสร้าง __init__

- ใช้กำหนดค่าเริ่มต้น (Constructor)
- self คือ object ที่ถูกสร้าง
- ควรใช้ type hinting เช่น name: str, age: int เพื่อความชัดเจน

☐ Inheritance

- Student สืบทอด Person ด้วย class Student(Person):
- ใช้ super() เพื่อเรียก constructor ของคลาสแม่
- สามารถ override method เช่น introduce()

☐ Polymorphism

- แม้ว่า Student จะมี method ชื่อเดียวกับ Person (introduce) แต่จะทำงานตามคลาสลูกเมื่อเรียกจาก object ของ Student

☐ เทคนิคขั้นสูง:

☐ Encapsulation

class BankAccount:

```
def __init__(self, owner, balance):
    self.owner = owner
    self.__balance = balance # private
```

```
def deposit(self, amount):
    if amount > 0:
        self.__balance += amount
```

```
def get_balance(self):
    return self.__balance
```

- ☐ __balance เป็นข้อมูลส่วนตัว ไม่สามารถเข้าถึงได้โดยตรงจากนอกคลาส

☐ ใช้ __str__ และ __repr__ (Magic Methods)

class Product:

```
def __init__(self, name, price):
    self.name = name
    self.price = price
```

```
def __str__(self):
    return f"{self.name} - {self.price:.2f} บาท"
```

```
def __repr__(self):
    return f"Product('{self.name}', {self.price})"
```

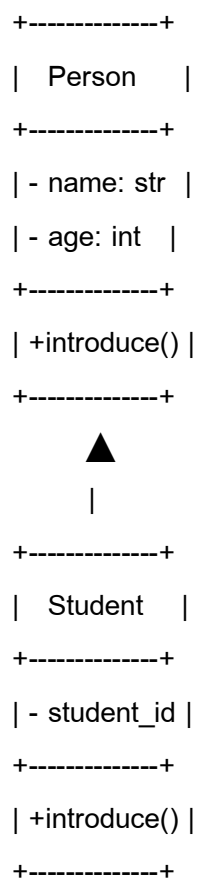
☐ แนวทางประยุกต์ใช้งานจริง

ตัวอย่างระบบ	คลาสที่เกี่ยวข้อง
ระบบจัดการนักเรียน	Person, Student, Teacher
ระบบร้านค้า	Product, Customer, Cart, Order
ระบบธนาคาร	Customer, BankAccount, Transaction

ต่อไปนี้เป็นตัวอย่างโปรแกรมการใช้ OOP ในภาษา Python จำนวน 10 ตัวอย่าง โดยแต่ละตัวอย่างมี

ครบ:

- ☐ UML Diagram
- ☐ โค้ด Python
- ☐ ผลลัพธ์ที่ได้จากการรัน
- ☐ คำอธิบายโค้ดแบบละเอียด

☐ ตัวอย่างที่ 1: Person และ Student (Inheritance + Override)☐ UML:☐ โค้ด:

```

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

    def introduce(self):
        return f'I'm {self.name}, {self.age} years old.'

class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)
        self.student_id = student_id

    def introduce(self):
        return f'I'm student {self.name}, ID: {self.student_id}'

```

```

p = Person("Alice", 45)
s = Student("Bob", 20, "S001")
print(p.introduce())
print(s.introduce())

```

☐ ผลรัน:

I'm Alice, 45 years old.

I'm student Bob, ID: S001

☐ อธิบายโค้ด:

- ใช้ `super().__init__()` เพื่อเรียก constructor ของคลาสแม่
- `introduce()` ถูก override ใน Student เพื่อให้แสดงเฉพาะข้อมูลของนักเรียน

☐ ตัวอย่างที่ 2: **BankAccount (Encapsulation)**

☐ UML:

```

+-----+
| BankAccount |
+-----+
| - __balance: float |
| - owner: str      |

```

```
+-----+
| +deposit()      |
| +get_balance()  |
+-----+
```

☐ โค้ด:

class BankAccount:

```
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance # private
```

```
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
```

```
    def get_balance(self):
        return self.__balance
```

```
acc = BankAccount("John", 1000)
acc.deposit(500)
print(acc.get_balance())
```

☐ ผลรัน:

1500

☐ อธิบายโค้ด:

- __balance เป็น attribute private
- การเข้าถึงข้อมูลภายในใช้ method get_balance() เพื่อป้องกันการแก้ค่าจากภายนอก

☐ ตัวอย่างที่ 3: Product และ Cart (Composition)

☐ UML:

```
+-----+      +-----+
| Product |<----->| Cart |
+-----+      +-----+
| - name   |      | - items |
| - price  |      +-----+
+-----+      | +add()  |
```

```
| +total() |
+-----+
```

☐ โค้ด:

```
class Product:
```

```
    def __init__(self, name, price):
        self.name = name
        self.price = price
```

```
class Cart:
```

```
    def __init__(self):
        self.items = []

    def add(self, product):
        self.items.append(product)

    def total(self):
        return sum(p.price for p in self.items)
```

```
cart = Cart()
cart.add(Product("Pen", 10))
cart.add(Product("Notebook", 50))
print("Total:", cart.total())
```

☐ ผลรัน:

```
Total: 60
```

☐ อธิบายโค้ด:

- สร้าง Product แล้วนำเข้า Cart
- ใช้ composition: Cart มีหลาย Product

ต่อไปนี้เป็นตัวอย่างโปรแกรม Python เชิง OOP พร้อม UML, ผลรัน, และ อธิบายโค้ด ตั้งแต่ ตัวอย่างที่ 4 - 10:

☐ ตัวอย่างที่ 4: Shape (Polymorphism)

☐ UML:

```
+-----+
```

```

| Shape |<<abstract>>
+-----+
| +area() |
+-----+
      ▲      ▲

```

```

+-----+ +-----+
| Rectangle | | Circle |
+-----+ +-----+
| width    | | radius   |
| height   | +-----+
+-----+ | +area()    |
| +area()  | +-----+
+-----+

```

□ โค้ด:

```
import math
```

```
class Shape:
```

```
    def area(self):
        raise NotImplementedError
```

```
class Rectangle(Shape):
```

```
    def __init__(self, w, h):
        self.w = w
        self.h = h
```

```
    def area(self):
        return self.w * self.h
```

```
class Circle(Shape):
```

```
    def __init__(self, r):
        self.r = r
```

```
    def area(self):
        return math.pi * self.r * self.r
```