บทที่ 8

แนะนำการเขียนโปรแกรมเชิงวัตถุ

ในบทนี้ผู้อ่านจะได้ทำความรู้จักและวิธีการเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming) หรือเรียกสั้น ๆ ว่า การเขียนโปรแกรมแบบ OOP ซึ่งเป็นรูปแบบการเขียนโปรแกรมได้รับความนิยมในปัจจุบัน แต่สำหรับผู้เริ่มฝึกเขียนโปรแกรมอาจจะมีความรู้สึกว่าทำความเข้าใจยากเพราะรูปแบบการเขียนโปรแกรมมีความซับซ้อน ดังนั้นจึงขอให้ผู้อ่านค่อย ๆ เริ่มศึกษาและทำความเข้าใจไปทีละขั้นตอน เนื่องจากมีคำศัพท์ที่ต้องทำความเข้าใจอยู่เป็นจำนวนมาก ถ้าหากเข้าใจหลักการเขียนโปรแกรมแบบ OOP แล้ว จะทำให้การพัฒนาโปรแกรมด้วยภาษาไพธอนที่มีความซับซ้อนกลายเป็นเรื่องง่าย ๆ แต่สำหรับในบทนี้เป็นเพียงการแนะนำให้รู้จักกับโปรแกรมแบบ OOP เท่านั้น ซึ่งอาจจะยังไม่ครอบคลุมเนื้อหาทั้งหมดของการเขียนโปรแกรมแบบ OOP เพราะมีเนื้อหาและรายละเอียดจำนวนมาก หากเนื้อหาในจุดใดทำให้ผู้อ่านเกิดความสงสัย สามารถศึกษาและค้นคว้าเพิ่มเติมจากแหล่งอื่น ๆ เช่น อินเตอร์เน็ต หนังสือ เป็นต้น ที่อธิบายถึงรูปแบบการเขียนโปรแกรม OOP ด้วยภาษาไพธอน

8.1 ทำไมต้องเขียนโปรแกรมแบบ OOP

วิธีการเขียนโปรแกรมเชิงโครงสร้าง (Structure Programming) หรือการเขียนโปรแกรมแบบเชิงฟังก์ชัน (Function Programming) เป็นรูปแบบการเขียนโปรแกรมแบบดั้งเดิมที่มีการใช้งานมานาน ถึงแม้ว่าจะช่วยแก้ไขปัญหาได้ในระดับหนึ่งและเหมาะสมกับผู้ที่ต้องการเริ่มศึกษาและเป็นพื้นฐานที่สำคัญสำหรับการเขียนโปรแกรม แต่เมื่อเจอปัญหาที่มีความซับซ้อนมาก ๆ จะพบว่าวิธีการเขียนโปรแกรมทั้ง 2 รูปแบบ ไม่สามารถตอบสนองต่อการแก้ไขปัญหาได้ รวมไปถึงการแก้ไขปรับปรุงคำสั่งโปรแกรมในอนาคต ทำให้เกิดวิธีการเขียนคำสั่งโปรแกรมรูปแบบใหม่ขึ้นมาคือ การเขียนโปรแกรมแบบ OOP ทำให้ช่วยแก้ไขปัญหาต่าง ๆ ข้างต้นได้ นอกจากนี้ยังสามารถช่วยให้เราประหยัดเวลาในการพัฒนาโปรแกรม รวมทั้งการบำรุงรักษาหรือการนำโปรแกรมกลับมาแก้ไขใหม่ และเพิ่มเติมส่วนต่าง ๆ ภายในโปรแกรมได้ง่ายขึ้น

8.2 เริ่มต้นเขียนโปรแกรมแบบ OOP ด้วยภาษาไพธอน

การพัฒนาโปรแกรมรูปแบบนี้จะมองทุกอย่างเป็นวัตถุ (Object) ไม่ว่าจะเป็นการเขียนโปรแกรมเกี่ยวกับการคำนวณหรือการเขียนโปรแกรมให้แสดงรายละเอียดสิ่งของต่าง ๆ เช่น หนังสือ สัตว์ รถยนต์ มอเตอร์ไซค์ จักรยาน โดยแต่ละวัตถุก็จะมีแอตทริบิวต์ (Attribute) เป็นการบ่งบอกถึงคุณสมบัติ (Property) ของวัตถุ และพฤติกรรม (Behavior) หรือเมธอด (Method) เป็นการบ่งบอกถึงวัตถุนั้นทำอะไรได้บ้าง ทั้งสองอย่างนี้ถูกบรรจุในรูปแบบการเขียนโปรแกรม OOP ผู้อ่านจะได้ทำความรู้จักและวิธีการนำมาใช้งานในหัวข้อต่อ ๆ ไป

8.2.1 การสร้างคลาส

สำหรับการเขียนโปรแกรมแบบ OOP จะมีการออกแบบโครงสร้างของออบเจ็คไว้ก่อนเสมอ เปรียบเสมือนการสร้างต้นแบบหรือพิมพ์เขียวขึ้นมา และมีการกำหนดองค์ประกอบคุณสมบัติและเมธอดต่าง ๆ ให้กับออบเจ็คนั้น ๆ โดยต้นแบบที่สร้างขึ้นมาในการเขียนโปรแกรมแบบ OOP จะเรียกว่า class (คลาส) ซึ่งต้นแบบที่สร้างขึ้นมานี้สามารถที่จะนำไปดัดแปลง แก้ไข เพิ่มเติมในภายหลังได้ หรือกล่าวได้อีกอย่างว่าการสร้างคลาสก็เพื่อจัดกลุ่มของโปรแกรมเชิงวัตถุที่มีลักษณะเหมือนกันให้อยู่ในกลุ่มคลาสเดียวกัน รูปแบบการสร้าง class แสดงได้ดังนี้

```
class className: # คำสั่งประกาศสร้างคลาส
    statement_1  # คำสั่งโปรแกรมที่ 1 ภายในคลาส
    ...
    ...
    ...
    statement_n  # คำสั่งโปรแกรมที่ n ภายในคลาส
```

ตามหลักการและโดยทั่วไปของการเขียนคำสั่งโปรแกรมแบบ OOP จะกำหนดให้ตัวอักษรตัวแรกของชื่อคลาสเป็นตัวพิมพ์ใหญ่เสมอ ยกตัวอย่างเช่น

```
class Book: # ประกาศสร้างคลาส Book
    statement(s)
```

```
class Dog: # ประกาศสร้างคลาส Dog
    statement(s)

class Student: # ประกาศสร้างคลาส Student
    statement(s)

class Calculate_area: # ประกาศสร้างคลาส Calculate_area
    statement(s)
```

8.2.2 การสร้างเมธอด

จุดประสงค์สำหรับการเมธอด (Method) ก็เพื่อให้ทำหน้าที่แสดงการกระทำอย่างใดอย่างหนึ่งของออบเจ็ค วิธีสร้างเมธอดขึ้นมาใช้งานก็มีหลากหลายรูปแบบตามจุดประสงค์ นอกจากนี้ยังมีแบบสำเร็จรูปที่ภาษาไพธอนเตรียมไว้ซึ่งจะนำมาใช้งานในคลาสได้เลย ต่อไปจะขอยกตัวอย่างการสร้างเมธอดแบบต่าง ๆ ที่นำขึ้นมาใช้งานในคลาสดังต่อไปนี้

Instance Method

การสร้างเมธอดขึ้นมาใช้งานเหมือนกับลักษณะวิธีการสร้างฟังก์ชันโดยทั่วไป แต่จะมีค่าพารามิเตอร์ชื่อ `self` กำกับไว้เป็นตัวแรกเสมอ ซึ่งเป็นการอ้างอิงถึงตัวเองและแอตทริบิวต์ตัวอื่น ๆ ที่อยู่ภายในเมธอด โดยจะถูกเรียกใช้งานผ่านทางอินสแตนซ์ (instance) ซึ่งจะกล่าวถึงในหัวข้อถัดไป หรือเรียกอีกอย่างว่าการสร้างออบเจ็คขึ้นมาก่อน หลังจากนั้นจึงใช้อินสแตนซ์ที่สร้างขึ้นมาอ้างอิงถึงเมธอดภายในคลาสและส่งผ่านค่าอาร์กิวเมนต์ที่มีค่าพารามิเตอร์คอยรับค่า หรือไม่มีก็ได้แต่ต้องมี Self ดังนี้

```
class Calculate_area:  # ประกาศสร้างคลาส
↪   Calculate_area
    def cal_rectangle(self, w, h): # สร้างเมธอดแบ
    ↪   บอินสแตนซ์คำนวณพื้นที่สี่เหลี่ยมผืนผ้าและกำหนด
    ↪   พารามิเตอร์
        return w * h # คำสั่งคืนค่ากลับจากการคำนวณ
```

Class Method

วิธีการสร้างเมธอดแบบนี้มีลักษณะเหมือนกับการสร้างเมธอดแบบอินสแตนซ์ แต่มีความแตกต่างกันตรงที่ค่าพารามิเตอร์ตัวแรกเปลี่ยนจาก `self` เป็น `cls` แทน และมีการกำหนด `@classmethod` ไว้ก่อนการสร้างเมธอดขึ้นมาใช้งาน และวิธีการเรียกใช้งานเมธอดแบบคลาสนี้ไม่ต้องมีการสร้างอินสแตนซ์สามารถเรียกผ่านคลาสโดยตรงได้เลย

```
class Calculate_area: # ประกาศสร้างคลาส
↪   Calculate_area
    @classmethod
    def cal_triangle(cls, b, h): # สร้างเมธอดแบบ
    ↪   คลาสคำนวณพื้นที่สามเหลี่ยมและกำหนดพารามิเตอร์
        return 0.5 * b * h # คำสั่งคืนค่ากลับจากการ
        ↪   คำนวณ
```

Static Method

เมธอดนี้มีการสร้างและการเรียกใช้งานแตกต่างจากทั้งสองวิธีคือ ไม่มีค่าพารามิเตอร์ตัวแรกกำกับไว้ แต่จะมีการกำหนด `@staticmethod` กำกับไว้ก่อนการสร้างเมธอดประเภทนี้

สำหรับเมธอดแบบสแตติกนี้จะมีการส่งผ่านค่าอาร์กิวเมนต์ไปให้ค่าพารามิเตอร์ได้โดยตรง เหมือนกับวิธีการสร้างฟังก์ชันแบบทั่วไปได้เลย และไม่มีการสร้างอินสแตนซ์สำหรับการอ้างอิงการเข้าถึงเมธอดแบบสแตติก

```
class Calculate_area: # ประกาศสร้างคลาส
↪   Calculate_area
    @staticmethod # กำหนดก่อนการสร้างเมธอดแบบสแตติก
    def cal_circle(r): # สร้างเมธอดสแตติกคำนวณพื้นที่
    ↪   วงกลมและกำหนดพารามิเตอร์
        return 3.14 * r * r # คำสั่งคืนค่ากลับจากการ
        ↪   คำนวณ
```

8.2.3 การสร้างอินสแตนซ์

การเรียกใช้งานคลาสเป็นการเข้าถึงข้อมูลที่อยู่ภายในคลาส ได้แก่ แอตทริบิวต์ และเมธอด ซึ่งเป็นการบ่งบอกว่ากำลังจะมีการสร้างออบเจ็คขึ้นมาใช้งาน ออบเจ็คที่สร้างขึ้นมาจะมีชื่อเรียกอีกอย่างว่า อินสแตนซ์ (Instance) มีรูปแบบการสร้างอินสแตนซ์ดังต่อไปนี้

```
instance_name = class_name():
```

`instance_name`	ชื่ออินสแตนซ์ที่ต้องการสร้าง
`class_name`	ชื่อคลาสที่ต้องการเข้าถึง

ตัวอย่าง 8.1
รูปแบบการสร้างอินสแตนซ์

```
class Calculate_area: # ประกาศสร้างคลาส Calculate_area
    def cal_rectangle(self, w, h):
        return w * h
    def cal_triangle(self, b, h):
        return 0.5 * b * h

cal = Calculate_area()   # สร้างอินสแตนซ์ของคลาส Calculae_area
```

หลังจากที่เราได้รู้จักกับคลาส เมธอด และอินสแตนซ์ไปแล้ว เราสามารถนำทั้ง 3 ส่วนนี้มาเขียนเป็นโปรแกรมแบบ OOP เบื้องต้นได้ โดยแสดงดังตัวอย่างต่อไปนี้ ซึ่งเป็นการสร้างโปรแกรมแบบ OOP คำนวณหาพื้นที่ โดยมี `Calculate_area` เป็นชื่อคลาส และประกอบด้วยเมธอด `cal_rectangle` เป็นเมธอดแบบอินสแตนซ์สังเกตได้จากค่าพารามิเตอร์ `self` ตัวแรกได้ถูกกำหนดไว้ เมื่อต้องการเรียกใช้งานต้องมีการสร้างอินสแตนซ์ขึ้นมาก่อน สำหรับเมธอด `cal_triangle` เป็นเมธอดแบบคลาสเนื่องจากมีค่าพารามิเตอร์ `cls` ได้ถูกกำหนดไว้เป็นตัวแรก และมี `@classmethod` กำหนดไว้ และเมธอดสุดท้ายเป็นเมธอดสแตติกซึ่งมี `@staticmethod` กำหนดไว้ การเรียกใช้งานทั้ง 3 เมธอดแสดงดังตัวอย่าง 8.2

ตัวอย่าง 8.2
การเขียนคำสั่งโปรแกรมแบบ OOP สำหรับคำนวณพื้นที่

```python
# ประกาศสร้างคลาส Calculate_area
class Calculate_area:
    # เมธอดแบบอินสแตนซ์
    def cal_rectangle(self, w, h):
        return w * h

    # เมธอดแบบคลาส
    @classmethod
    def cal_triangle(cls, b, h):
        return 0.5 * b * h

    # เมธอดแบบสแตติก
    @staticmethod
    def cal_circle(r):
        return 3.14 * r * r

# สร้างอินสแตนซ์คลาส Calculate_area
cal = Calculate_area()

# เรียกใช้เมธอด cal_rectangle พร้อมส่งค่าอาร์กิวเมนต์
cal_rec = cal.cal_rectangle(3, 5)

# เรียกใช้เมธอด cal_triangle พร้อมส่งค่าอาร์กิวเมนต์
cal_tri = Calculate_area.cal_triangle(6, 7)

# เรียกใช้เมธอด cal_circle พร้อมส่งค่าอาร์กิวเมนต์
cal_circle = Calculate_area.cal_circle(5)

# แสดงผลการคำนวณของเมธอด cal_rectangle
print('พื้นที่สี่เหลี่ยมผืนผ้า', cal_rec)

# แสดงผลการคำนวณของเมธอด cal_triangle
print('พื้นที่สามเหลี่ยม', cal_tri)

# แสดงผลการคำนวณของเมธอด cal_circle
print('พื้นที่วงกลม', cal_circle)
```

```
พื้นที่สี่เหลี่ยมผืนผ้า 15
พื้นที่สามเหลี่ยม 21.0
พื้นที่วงกลม 78.5
```

จากตัวอย่าง 8.2 เป็นการเขียนคำสั่งโปรแกรมแบบ OOP คำนวณพื้นที่ โดยนำเสนอตัวอย่างการใช้เมธอดแบบต่าง ๆ กัน

บรรทัดที่ 4-5	เป็นการสร้างเมธอดอินสแตนซ์ มีพารามิเตอร์ตัวแรกเป็น `self` เสมอ ส่วนพารามิเตอร์ที่เหลือเป็นพารามิเตอร์คอยรับค่าที่ถูกส่งเข้ามาประมวลผล
บรรทัดที่ 8-10	เป็นการสร้างเมธอดคลาส สังเกตได้จากมี `@classmethod` กำกับไว้ก่อนการสร้างเมธอด และมีพารามิเตอร์ตัวแรกเป็น `cls` เสมอ ส่วนพารามิเตอร์ที่เหลือเป็นพารามิเตอร์คอยรับค่าที่ถูกส่งเข้ามาประมวลผล
บรรทัดที่ 13-15	เป็นการสร้างเมธอดสเตติกซึ่งจะมี `@staticmethod` กำกับไว้ และมีพารามิเตอร์คอยรับค่านำไปประมวลผลต่อเท่านั้น จะไม่มีพารามิเตอร์ `self` หรือ `cls` เหมือนกับเมธอด 2 แบบแรก
บรรทัดที่ 18	สร้างอินสแตนซ์ชื่อ `cal` จากคลาส `Calculate_area` เอาไว้ใช้อ้างอิงการเข้าถึงเมธอดแบบอินสแตนซ์
บรรทัดที่ 21	ส่งค่าข้อมูลให้กับเมธอดแบบอินสแตนซ์ และผลลัพธ์ที่ได้จะถูกเก็บไว้ที่ตัวแปร `cal_rec`
บรรทัดที่ 24	ส่งค่าข้อมูลให้กับเมธอดคลาส โดยการใช้ชื่อคลาสอ้างอิงในการเข้าถึง และผลลัพธ์ที่ได้จากการประมวลผลจะถูกเก็บไว้ที่ตัวแปร `cal_tri`
บรรทัดที่ 27	ส่งค่าข้อมูลให้กับเมธอดแบบสแตติก โดยการใช้ชื่อคลาสอ้างอิงในการเข้าถึง และผลลัพธ์ที่ได้จะถูกเก็บไว้ที่ตัวแปร `cal_circle`
บรรทัดที่ 30, 33, 36	แสดงผลลัพธ์ที่ได้จากการประมวลของเมธอดต่าง ๆ ที่ถูกส่งกลับคืนมา

8.3 การสร้างแอตทริบิวต์ และคอนสตรัคเตอร์

การสร้างแอตทริบิวต์ของคลาส (Class Attribute) ก็เปรียบเสมือนการสร้างตัวแปรขึ้นมาใช้งาน หรือกล่าวได้ว่าเป็นการกำหนดคุณสมบัติของวัตถุในเบื้องต้นที่จะถูกนำไปใช้ร่วมกับเมธอดอื่น ๆ ภายในคลาส โดยจะมีการกำหนดค่าให้กับแอตทริบิวต์หรือไม่มีการกำหนดค่าไว้ก็ได้ มีรูปแบบการสร้างดังแสดงตัวอย่าง โดยที่ `wide, high` และ `isbn` คือ แอตทริบิวต์ของคลาส `Book`

ตัวอย่าง 8.3
แสดงข้อมูลในแอตทริบิวต์ของคลาส

```python
# ประกาศสร้างคลาส Book
class Book:
    # สร้างแอตทริบิวต์ของคลาสและกำหนดค่า
    wide = 25
    high = 30
    page = 250

    # สร้างเมธอดแบบอินสแตนซ์และพารามิเตอร์รับค่า
    def showBook(self, name, isbn):
        self.isbn = isbn
        self.name = name

        # คืนค่ากลับแสดงผลลัพธ์
        return self.name, Book.wide, Book.high, Book.page,
        ↪  self.isbn


# สร้างอินสแตนซ์จากคลาส Book()
b_IT = Book()

# สร้างอินสแตนซ์จากคลาส Book()
b_Web = Book()

# แสดงผลลัพธ์จากการสร้างอินสแตนส์อ้างอิงไปยังเมธอด showBook() พร้อมทั้งส่งค่าอากิวเมนต์
print(b_IT.showBook('Python Programming', '123-456-789-0'))
print(b_Web.showBook('Web Programming', '777-000-222-0'))
```

```
('Python Programming', 25,30,250, '123-456-789-0')
('Web Programming', 25,30,250,'777-000-222-0')
```

จากตัวอย่างโปรแกรมเราจะเห็นว่ามีการสร้างแอตทริบิวต์ของคคลาสขึ้นมาไว้ใช้งาน โดยปกติแล้วแอตทริบิวต์ของคลาสจะถูกสร้างขึ้นมาและกำหนดค่าเบื้องต้นไว้เลยเพื่อความสะดวก

ในการนำไปใช้งานร่วมกับเมธอดอื่น ๆ ที่อยู่ภายในคลาส นอกจากนี้ภายในคำสั่งโปรแกรมมีการสร้างเมธอด `showBook()` สำหรับแสดงข้อมูลบางส่วนของหนังสือ โดยการนำเอาค่าแอตทริบิวต์ของคลาสมาแสดงผล และมีการสร้างพารามิเตอร์คอยรับค่าไว้อีกสองตัว

คอนสตรัคเตอร์ (Constructor) คือ เมธอดประเภทหนึ่งที่ถูกสร้างขึ้นมาใช้งาน และภายในเมธอดมีการสร้างแอตทริบิวต์เพื่อเป็นการกำหนดข้อมูลเบื้องต้นของวัตถุหรืออินสแตนซ์ที่จะถูกสร้างขึ้น การสร้างคอนสตรัคเตอร์จะเรียกใช้งานเมธอด `__init__()` (เครื่องหมายที่อยู่ด้านหน้าและหลังคือ Double underscore หรือเรียกอีกอย่างว่า dunder) เมื่อมีการสร้างคอนสตรัคเตอร์ขึ้นมาใช้งาน จะทำให้คอนสตรัคเตอร์มีการทำงานอัตโนมัติทุกครั้งเมื่อมีการสร้างอินแตนซ์จากคลาส คอนสตรัคเตอร์ที่ถูกสร้างขึ้นจะมีค่าพารามิเตอร์ไว้คอยรับค่าหรือไม่มีก็ได้ แต่ค่าพารามิเตอร์ที่ต้องมีเสมอคือ self และต้องกำหนดเป็นตัวแรก

การสร้างคอนสตรัคเตอร์แบบมีพารามิเตอร์รับค่า

```
class className:
    # สร้างเมธอดคอนสตรัคเตอร์
    def __init__(self, param1, pram2,
    ↪  param3,...,param_n)
        self.param1 = param1
        self.param2 = param2
        self.param3 = param3
        ...
        self.param_n = param_n
```

การสร้างคอนสตรัคเตอร์แบบไม่มีพารามิเตอร์รับค่า

```
class className:
    # สร้างเมธอดคอนสตรัคเตอร์
    def __init__(self)
        self.attribute1 = ''
        self.attribute2 = ''
        self.attribute3 = ''
        ...
        self.attribute_n = ''
```

ตัวอย่าง 8.4
การสร้างคอนสตรัคเตอร์แบบมีค่าพารามิเตอร์คอยรับค่า

```python
# ประกาศสร้างคลาส Book
class Book:
    # สร้างคอนสตรัคเตอร์แบบมีการรับค่าพารามิเตอร์
    def __init__(self, name, wide, high, page, isbn, price):
        # สร้างแอตทริบิวต์
        self.name = name
        self.wide = wide
        self.high = high
        self.page = page
        self.isbn = isbn
        self.price = price

    def showBook(self):  # สร้างเมธอดแบบอินสแตนซ์คืนค่ากลับแสดงผลข้อมูล
        return self.name, self.wide, self.high, self.page,
        ↪  self.isbn, self.price


# สร้างอินสแตนซ์ b_IT พร้อมส่งค่า
b_IT = Book('Python Programming', 25, 30, 350, '123-456-789-0',
↪  300)
print(b_IT.showBook())  # แสดงผลลัพธ์จากอินสแตนซ์ b_IT
b_IT.price = 350  # กำหนดราคาหนังสือให้กับอินสแตนซ์ b_IT ใหม่
print(b_IT.showBook())  # แสดงผลลัพธ์จากอินสแตนซ์ b_IT อีกครั้ง

# สร้างอินสแตนซ์ b_Web พร้อมส่งค่า
b_Web = Book('Web Programming', 20, 25, 300, '777-000-222-0',
↪  375)
print(b_Web.showBook())  # แสดงผลลัพธ์จากอินสแตนซ์ b_Web'
```

```
('Python Programming', 25,30,350, '123-456-789-0', 300)
('Python Programming', 25,30,350, '123-456-789-0', 350)
('Web Programming', 20,25,30,300,'777-000-222-0',375)
```

จากตัวอย่าง 8.4 เป็นการเขียนคำสั่งโปรแกรมสร้างคอนสตรัคเตอร์แบบมีค่าพารามิเตอร์คอยรับค่าข้อมูล เมื่อนำค่าพารามิเตอร์ไปสร้างแอตทริบิวต์ในบรรทัดที่ 6-11 จะมี `self` นำหน้าทุกค่าพารามิเตอร์ก่อนไปใช้งาน เมื่อเรียกใช้คลาสทำให้เมธอด `__init__()` ทำงานอัตโนมัติ ทำให้เราส่งข้อมูลเข้าไปยังคลาสได้เลยในขณะที่กำลังมีการสร้างอินสแตนซ์ในบรรทัดที่ 18 และเมื่อต้องการนำค่าข้อมูลที่ถูกเก็บอยู่ในแอตทริบิวต์ตัวใดมาแสดงผล หรือต้องการเปลี่ยนค่าข้อมูลของแอตทริบิวต์ตัวใด ให้ใช้ชื่ออินสแตนซ์ที่สร้างขึ้นมาตามด้วยเครื่องหมาย `(.)` แล้วตามด้วยชื่อแอตทริบิวต์ที่ต้องการนำมาแสดงผลหรือต้องการเปลี่ยนค่าข้อมูล แสดงดังตัวอย่างในบรรทัดที่ 20 เป็นการเปลี่ยนราคาหนังสือให้กับอินสแตนซ์ `b_IT` ใหม่ เห็นได้จากผลลัพธ์ที่ถูกแสดงออกมา

ตัวอย่าง 8.5
การสร้างคอนสตรัคเตอร์แบบไม่มีค่าพารามิเตอร์คอยรับค่า

```
# ประกาศสร้างคลาส Book
class Book:
    # สร้างคอนสตรัคเตอร์แบบไม่มีการรับค่าพารามิเตอร์
    def __init__(self):
        self.name = ''
        self.wide = ''
        self.high = ''
        self.page = ''
        self.isbn = ''
        self.price = ''

    # สร้างเมธอดแบบอินสแตนซ์คืนค่ากลับแสดงผลข้อมูล
    def showBook(self):
        return self.name, self.wide, self.high, self.page,
        ↪  self.isbn, self.price


b_Linux = Book()  # สร้างอินสแตนซ์ b_Linux
# กำหนดค่าข้อมูลให้แต่ละแอตทริบิวต์ของอินสแตนซ์ b_IT
b_Linux.name = 'Linux Programming'
b_Linux.wide = 25
b_Linux.high = 30
b_Linux.page = 350
b_Linux.isbn = '444-454-799-0'
b_Linux.price = 300

b_DB = Book()  # สร้างอินสแตนซ์ b_DB
# กำหนดค่าข้อมูลให้แต่ละแอตทริบิวต์ของอินสแตนซ์ b_DB
b_DB.name = 'Database Programming'
b_DB.wide = 30
b_DB.high = 35
b_DB.page = 377
b_DB.isbn = '444-333-222-0'
b_DB.price = 350
# แสดงผลลัพธ์จากอินสแตนซ์ b_Linux และ b_DB โดยอ้างอิงถึงเมดธอดแบบอินสแตนซ์
↪  showBook()
print(b_Linux.showBook())
print(b_DB.showBook())
```

```
('Linux Python Programming', 25,30,350, '444-454-799-0', 300)
('Database Programming', 30,35,377, '444-333-222-0', 350)
```

จากตัวอย่างที่ 8.5 เป็นการสร้างคอนสตรัคเตอร์ที่ไม่มีพารามิเตอร์คอยรับค่า แต่จะมีเฉพาะแอตทริบิวต์ที่สร้างขึ้นมาไว้รอรับค่าที่จะถูกกำหนดจากการสร้างอินสแตนซ์ไว้ 2 ตัว ได้แก่ `b_IT` และ `b_DB` แสดงในบรรทัดที่ 17 และ 26 เมื่อต้องการกำหนดค่าแอตทริบิวต์ให้เราใช้ชื่ออินสแตนซ์ที่สร้างขึ้นมาตามด้วยเครื่องหมาย `(.)` แล้วตามด้วยชื่อแอตทริบิวต์ที่ต้องการกำหนดค่า สำหรับการนำข้อมูลออกมาแสดงผลก็ให้อ้างถึงเมธอดคลาสผ่านทางอินสแตนซ์แต่ละตัว

8.4 การสืบทอด

การสืบทอด (Inheritance) เป็นหลักการสำคัญในการเขียนโปรแกรมเชิงวัตถุ กล่าวคือ คุณสมบัติต่าง ๆ (แอตทริบิวต์) ที่สร้างขึ้นภายในคลาสสามารถที่จะถูกสืบทอด (inherit) ไปยังคลาสอื่น ๆ ได้ เมื่อสร้างคลาสใหม่ขึ้นมาใช้งาน และภายในคลาสใหม่อาจจะมีแอตทริบิวต์หรือเมธอดที่เหมือนกับคลาสที่สร้างขึ้นมาใช้งานก่อนหน้านี้ หากต้องสร้างแอตทริบิวต์หรือเมธอดขึ้นมาใหม่อีกก็จะทำให้เสียเวลาในการเขียนคำสั่งโปรแกรมและเกิดความซ้ำซ้อน การสืบทอดจากคลาสหนึ่งไปยังอีกคลาสหนึ่งช่วยทำให้เราประหยัดเวลาในการเขียนคำสั่งโปรแกรม เพราะไม่จำเป็นต้องสร้างแอตทริบิวต์หรือเมธอดที่มีอยู่ในคลาสเดิมคลาสขึ้นมาอีก โดยคลาสที่ถูกสืบทอดเราเรียกว่า คลาสแม่ (Superclass หรือ Parent Class) สำหรับคลาสที่ได้รับสืบทอดเรียกว่า คลาสลูก (Subclass) นอกจากนี้ยังสามารถสืบทอดได้หลายคลาส

รูปแบบการสร้างคลาสลูกให้สืบทอดจากคลาสแม่

```
class Superclass:  # สร้างคลาสแม่
    Statement(s)

class Subclass(Superclass):  # สร้างคลาสลูกสืบทอดจาก
↪   คลาสแม่
    Statement(s)
```

ตัวอย่าง 8.6
ลักษณะการสืบทอดคลาส

```python
class Book:  # ประกาศสร้างคลาส Book เป็นคลาสแม่

    def __init__(self, name, size, page, price):  # คอนสตรัคเตอร์ กำหนดแอตทริบิวต์
        self.name = name
        self.size = size
        self.page = page
        self.price = price

    def showBook(self):  # สร้างเมธอดแบบอินสแตนซ์สำหรับแสดงผลข้อมูล
        return self.name, self.size, self.page, self.price


class Book_IT(Book):  # สร้างคลาส Book_IT เป็นคลาสลูกสืบทอดมาจากคลาส Book
    pass


# กำหนดข้อมูลให้กับอินสแตนซ์คลาสแม่
book = Book('Python Programming', '25 x 30', 275, 300)
# กำหนดข้อมูลให้กับอินสแตนซ์คลาสลูก
b_IT = Book_IT('Information Technology', '25 x 32', 390, 250)
# แสดงผลข้อมูลอ้างอิงจากอินสแตนซ์คลาสแม่ไปยังเมธอด showBook() ของคลาสแม่
print(book.showBook())
# แสดงผลข้อมูลอ้างอิงจากอินสแตนซ์คลาสลูกไปยังเมธอด showBook() ของคลาสแม่
print(b_IT.showBook())
```

```
('Python Programming', 25 x 30, 275, 300,
('Information Technology', 25 x 32, 390, 250)
```

จากตัวอย่าง 8.6 เป็นการสาธิตการเขียนคำสั่งโปรแกรมให้มีการสืบทอดจากคลาส Book ซึ่งเป็นคลาสแม่ ไปยังคลาส Book_IT ซึ่งเป็นคลาสลูก โดยที่ภายในคลาส Book_IT มี

เฉพาะคำสั่ง `pass` เท่านั้น เมื่อสร้างอินสแตนซ์ในบรรทัดที่ 20 ขึ้นมาแล้วเราสามารถที่จะเข้าถึงคอนสตรัคเตอร์ของคลาสแม่ได้เลย และเรียกใช้งานเมธอดที่มีอยู่ในคลาสแม่ได้

ตัวอย่าง 8.7
การสืบทอดคุณสมบัติคลาสแม่ไปยังคลาสลูก

```
class Book:
    def __init__(self, name, size, page, price):
        self.name = name
        self.size = size
        self.page = page
        self.price = price

    def showBook(self):
        return self.name, self.size, self.page, self.price


class Book_IT(Book):
    def __init__(self, name, size, page, price, isbn,
    ↪  publisher):
        super().__init__(name, size, page, price)
        self.isbn = isbn
        self.publisher = publisher

    def showBook_IT(self):
        return super().showBook(), self.isbn, self.publisher

book = Book('Python Programming', '25 x 30', 275, 300)
b_IT = Book_IT('Information Technology', '25 x 32', 390, 250,
↪  '123-456-789-0', 'ABC')
print(book.showBook())
print(b_IT.showBook_IT())'
```

```
('Python Programming', '25 x 30', 275, 300)
(('Information Technology', '25 x 32', 390 ,250),
↪  '123-456-789-0', 'ABC'
```

จากตัวอย่าง 8.7 ได้เขียนคำสั่งโปรแกรมเพิ่มเติมภายในคลาส Book_IT ซึ่งเป็นคลาสลูกสืบทอดมาจากคลาส Book ซึ่งเป็นคลาสแม่

บรรทัดที่ 13 สร้างคอนสตรัคเตอร์เก็บแอตทริบิวต์ของคลาสลูกที่สร้างขึ้นมาและมีการสืบทอดคุณสมบัติมาจากคลาสแม่ ภายในคอนสตรัคเตอร์ของคลาส `Book_IT` มีพารามิเตอร์เพิ่มเข้ามาอีก 2 ตัว คือ `isbn` และ `publisher` ส่วนที่เหลือคือพารามิเตอร์ที่ต้องสืบทอดมาจากคลาสแม่

บรรทัดที่ 14 ใช้คำสั่ง `super()__init__(parameter(s))` สร้างแอตทริบิวต์ที่สืบทอดมาจากคลาสแม่ จากนั้นสร้างแอตทริบิวต์ที่เพิ่มเข้ามาในคลาสลูก

บรรทัดที่ 18 เป็นการสร้างเมธอด `showBook_IT()` ซึ่งเป็นเมธอดแบบอินสแตนซ์ สำหรับส่งค่ากลับมาแสดงผลเมื่อมีการเรียกใช้งาน

บรรทัดที่ 19 ใช้คำสั่ง `super().method()` ที่ต้องการสืบทอด จากตัวอย่างคือเมธอด `showBook()` และเพิ่มแอตทริบิวต์ส่วนที่เหลือ

บรรทัดที่ 24 เปลี่ยนจากการอ้างอิงผ่านอินสแตนซ์จากเมธอด `showBook()` ซึ่งเป็นของคลาสแม่ เป็นเมธอด `showBook_IT()` ซึ่งเป็นของคลาสลูกแทน

ตัวอย่าง 8.8
การสืบทอดคุณสมบัติคลาสแม่ไปยังคลาสลูก

```
1  class Book:
2      def __init__(self, name, size, page, price):
3          self.name = name
4          self.size = size
5          self.page = page
6          self.price = price
7
8      def showBook(self):
9          return self.name, self.size, self.page, self.price
10
11
```

```
class Book_IT(Book):
    def __init__(self, name, size, page, price, isbn,
    ↪ publisher):
        super().__init__(name, size, page, price)
        self.isbn = isbn
        self.publisher = publisher

    def showBook_IT(self):
        return super().showBook(), self.isbn, self.publisher


class BookType(Book_IT):
    def __init__(self, name, size, page, price, isbn, type,
    ↪ publisher):
        super().__init__(name, size, page, price, isbn,
        ↪ publisher)
        self.type = type

    def showBookType(self):
        return super().showBook_IT(), self.type


book = Book('Python Programming', '25 x 30', 275, 300)
b_IT = Book_IT('Information Technology', '25 x 32', 390, 250,
↪ '123-456-789-0', 'ABC')
b_IT1 = BookType('Information Technology', '25 x 32', 390, 250,
↪ '123-456-789-0', 'Computer', 'ABC')
print(book.showBook())
print(b_IT.showBook_IT())
print(b_IT1.showBookType())
```

```
('Python Programming', '25 x 30', 275, 300)
(('Information Technology', '25 x 32', 390, 250),
↪ '123-456-789-0', 'ABC')
((('Information Technology', '25 x 32', 390, 250),
↪ '123-456-789-0', 'ABC'), 'Computer')
```

จากตัวอย่าง 8.8 ได้สร้างคลาสลูกขึ้นมาอีก 1 คลาสคือ BookType และเพิ่มแอตทริบิวต์อีก 1 ตัวคือ type เพื่อบอกว่าหนังสือเล่มนี้เป็นหนังสือประเภทอะไร โดยคลาสนี้จะสืบทอดคุณสมบัติมาจากคลาส Book_IT ซึ่งเป็นคลาสแม่ สำหรับการเพิ่มคอนสตรัคเตอร์จะเหมือนกับตัวอย่าง 8.7 ส่วนผลลัพธ์ให้เพิ่มข้อมูลดังตัวอย่างในบรรทัดที่ 21 และแสดงผลลัพธ์โดยอ้างอิงจากอินสแตนซ์ที่ชื่อ b_IT1 ไปยังเมธอด showBookType() ตามตัวอย่างในบรรทัดที่ 25

8.5 เมธอด __str__() และ __repr__()

การจัดรูปแบบการแสดงผลข้อมูลก็เป็นอีกส่วนหนึ่งที่มีความสำคัญ ภาษาไพธอนได้จัดเตรียมเมธอดที่ใช้สำหรับแสดงผลลัพธ์ที่คืนค่าจากคลาสที่ถูกอินสแตนซ์เรียกใช้งานในบางกรณี ผลลัพธ์ที่แสดงออกมาจะกลายเป็นตำแหน่งข้อมูลที่อยู่ในหน่วยความจำแทน แต่เราสามารถใช้เมธอด __str__() และ __repr__() เข้ามาช่วยจัดรูปแบบการแสดงผลข้อมูลได้ แต่ทั้งสองเมธอดนี้มีผลลัพธ์ที่แสดงออกมาแตกต่างกันดังตัวอย่างต่อไปนี้

ตัวอย่าง 8.9
การใช้เมธอด `__str__()` แสดงผลลัพธ์

```
1  class StudentTest:
2      def __init__(self, name, score1, score2, score3):
3          self.name = name
4          self.score1 = score1
5          self.score2 = score2
6          self.score3 = score3
7
8      def sumScore(self):  # เมธอดแบบอินสแตนซ์
9          return self.score1 + self.score2 + self.score3
10
11     def __str__(self):   # เมธอดคืนค่ากลับแสดงผลข้อมูล
12         return f'Name: {self.name}, Total of score:
           ↪  {self.sumScore()}'
13
14
15 std1 = StudentTest('Janis', 20, 35, 25)
16 print(std1.name, std1.sumScore())
17 print(std1)
```

```
Jantra 80
Name: Janis, Total of score: 80
```

จากตัวอย่าง 8.9 ในบรรทัดที่ 11 แสดงการใช้เมธอด `__str__()` และมีการจัดรูปแบบข้อมูล และทำการคืนค่ากลับในบรรทัดที่ 12 ด้วยคำสั่ง **`return`** โดยมีการใช้ f-string เข้าช่วยในการจัดรูปแบบการแสดงผล เมื่อสังเกตคำสั่งแสดงผลลัพธ์ (`print`) ในบรรทัดที่ 16 และ 17 จะมีวิธีการเขียนคำสั่งที่แตกต่างกัน โดยในบรรทัดที่ 15 จะใช้อินสแตนซ์ในการเข้าถึงแอตทริบิวต์ที่อยู่ในคลาสถึงจะนำข้อมูลกลับมาแสดงผลลัพธ์ได้ แต่ในบรรทัดที่ 16 จะใช้เพียงอินสแตนซ์ที่สร้างขึ้นมาเท่านั้นก็สามารถแสดงผลลัพธ์ได้แล้ว เนื่องจากเมธอด `__str__()` เป็นตัวส่งข้อมูลกลับมาให้

ตัวอย่าง 8.10
การใช้เมธอด `__repr__()` แสดงผลลัพธ์

```
1  class StudentTest:
2      def __init__(self, name, score1, score2, score3):
3          self.name = name
4          self.score1 = score1
5          self.score2 = score2
6          self.score3 = score3
7
8      def sumScore(self):
9          return self.score1 + self.score2 + self.score3
10
11     def __repr__(self):  # เมธอดคืนค่ากลับแสดงผลข้อมูล
12         return repr((self.name, self.sumScore()))
13
14
15 std1 = StudentTest('Janis', 20, 35, 25)
16 print(std1.name, std1.sumScore())
17 print(std1)
```

```
Janis 80
('Janis', 80)
```

เมื่อใช้เมธอด `__repr__()` คืนค่ากลับมาแสดงผล ตามตัวอย่างบรรทัดที่ 11-12 ผลลัพธ์ที่ออกมาจะเป็นชนิดข้อมูล และเมื่อมีชนิดข้อมูลสตริงปรากฏอยู่ภายในจะมีเครื่องหมาย `(' ')` ครอบไว้ ถึงแม้ว่าจะเป็นชนิดข้อมูลสตริงก็ตาม อย่างไรก็ตามเราสามารถปรับเปลี่ยนวิธีการแสดงผลโดยใช้เมธอด `__repr__()` ได้หลากหลายรูปแบบ ให้ผู้อ่านทดลองแก้ไขคำสั่งรูปแบบการคืนค่ากลับมาแสดงผล และทดสอบการทำงานคำสั่งของโปรแกรมหลาย ๆ แบบ เพื่อให้เกิดความเข้าใจการทำงานของทั้งเมธอดและนำไปใช้งานในการเขียนโปรแกรมแบบ OOP

8.6 การโอเวอร์โหลดตัวดำเนินการ

เมื่อสร้างอินสแตนซ์ (วัตถุ) ขึ้นมาจากคลาสใด ๆ แล้ว และเมื่อต้องการดำเนินการอย่างใดอย่างหนึ่งระหว่างอินสแตซ์ เช่น การบวก การลบ การหาร เป็นต้น เครื่องหมายตัวดำเนินการต่าง ๆ จะไม่สามารถนำมาใช้งานได้ตรง ๆ กับอินสแตนซ์ แต่มีวิธีการดำเนินการระหว่างอินสแตนซ์ด้วยเครื่องหมาย เช่น บวก ลบ คูณ หาร เป็นต้น เหล่านี้จะทำได้ด้วยการทำโอเวอร์โหลดตัวดำเนินการ (Operator Overloading) โดยเรียกใช้งานเมธอดพิเศษ เช่น เมธอด `__add__()` ใช้สำหรับโอเวอร์โหลดเครื่องหมายบวก เมธอด `__sub__()` ใช้สำหรับโอเวอร์โหลดเครื่องหมายลบ เมธอด `__mul__()` ใช้สำหรับโอเวอร์เครื่องหมายคูณ เป็นต้น

ตัวอย่าง 8.11
การโอเวอร์โหลดตัวดำเนินการบวก

```
class StudentTest:
    def __init__(self, score):
        self.score = score

    def __add__(self, other):  # เมธอดโอเวอร์โหลดตัวดำเนินการบวก
        result = self.score + other.score
        return StudentTest(result)

    def __str__(self):  # เมธอดจัดรูปแบบชนิดข้อมูลสตริง
        return 'Total of score = {}'.format(self.score)


score1 = StudentTest(20)
score2 = StudentTest(30)
score3 = StudentTest(25)
sumScore = score1 + score2 + score3
print(sumScore)
```

```
Total of score = 75
```

จากตัวอย่าง 8.11 เมื่อต้องการบวกคะแนนของแต่ละอินสแตนซ์ จะมีการสร้างเมธอด `__add__()` ในบรรทัดที่ 5 ขึ้นมาทำการโอเวอร์โหลดเครื่องหมายบวก สำหรับเมธอด `__str__()` เป็นเมธอดที่ใช้สำหรับจัดรูปแบบชนิดข้อมูลสตริง หากเราไม่สร้างเมธอดนี้ขึ้นมา ผลลัพธ์ที่ได้จะเป็นค่าตำแหน่งข้อมูลที่เก็บอยู่ในหน่วยความจำแทน

ตัวอย่าง 8.12
การโอเวอร์โหลดตัวดำเนินการเปรียบเทียบ

```
class Compare:
    def __init__(self, x):
        self.x = x

    def __ge__(self, other):  # เมธอดโอเวอร์โหลดเครื่องหมายมากกว่าหรือเท่ากับ
        if (self.x >= other.x):
            text = 'num1 is greater than or equal num2'
        else:
            text = 'num1 is less than num2'
        return text

    def __ne__(self, other):  # เมธอดโอเวอร์โหลดเครื่องหมายไม่เท่ากับ
        if (self.x != other.x):
            text = 'Both are not equal'
        else:
            text = 'Both are equal'
        return text


num1 = Compare(200)
num2 = Compare(450)
result_n = num1 >= num2
print('The result of comparison is = ', result_n)

str1 = Compare('Python')
str2 = Compare('C++')
result_s = str1 == str2
print('The result of comparison is = ', result_s)
```

```
The result of comparison is =  num1 is less than num2
The result of comparison is =  False
```

สัญลักษณ์	ชื่อเมธอด
`+`	`__add__(self, other)`
`-`	`__sub__(self, other)`
`*`	`__mul__(self, other)`
`/`	`__truediv__(self, other)`
`//`	`__floordiv__(self, other)`
`%`	`__mod__(self, other)`
`**`	`__pow__(self, other)`

ตาราง 8.1: เมธอดโอเวอร์โหลดตัวดำเนินการ

ยังมีเมธอดอื่น ๆ อีกจำนวนมากที่นำมาใช้สำหรับโอเวอร์โหลดตัวดำเนินการและเครื่องหมายต่าง ๆ ในภาษาไพธอน จึงขอยกตัวอย่างเมธอดที่น่าจะได้นำมาใช้งานบ่อย ๆ ในการเขียนโปรแกรมแบบ OOP ดังแสดงในตารางต่อไปนี้

สัญลักษณ์	ชื่อเมธอด
`<`	`__lt__(self, other)`
`>`	`__gt__(self, other)`
`<=`	`__le__(self, other)`
`>=`	`__ge__(self, other)`
`==`	`__eq__(self, other)`
`!=`	`__ne__(self, other)`

ตาราง 8.2: เมธอดโอเวอร์โหลดตัวดำเนินการเปรียบเทียบ

สัญลักษณ์	ชื่อเมธอด
`-=`	`__isub__(self, other)`
`+=`	`__iadd__(self, other)`
`*=`	`__imul__(self, other)`
`**=`	`__ipow__(self, other)`
`/=`	`__idiv__(self, other)`
`//=`	`__ifloordiv__(self, other)`
`%=`	`__imod__(self, other)`

ตาราง 8.3: เมธอดโอเวอร์โหลดการกำหนดค่า

8.7 การโอเวอร์โหลดดิ้งเมดธอด

เมื่อคลาสลูกสืบทอดมาจากคลาสแม่ ทำให้คลาสลูกมีคุณสมบัติและมีพฤติกรรมเหมือนกับคลาสแม่ทุกประการ แต่ในบางกรณีคลาสลูกอาจจะมีพฤติกรรมที่ต่างไปจากคลาสแม่ก็ได้ การโอเวอร์โหลดดิ้งเมธอด (Method Overloading) เป็นการกำหนดพฤติกรรมของคลาสลูกให้ต่างจากคลาสแม่ แต่มีชื่อเมธอดที่เหมือนกัน ยกตัวอย่างเช่น เราสร้างคลาสโทรศัพท์มือถือแล้วกำหนดให้เป็นคลาสแม่ และมีเมธอดเปิด-ปิดหน้าจอด้วยวิธีการสไลด์หน้าจอ และสร้างคลาสลูกขึ้นมาสืบทอดและมีชื่อเมธอดที่เหมือนกันกับคลาสแม่ แต่มีวิธีการเปิด-ปิดหน้าจอที่ต่างกัน เช่น สแกนลายนิ้วมือ สแกนใบหน้า ป้อนรหัสก่อนเข้าใช้งานโทรศัพท์ เป็นต้น

ตัวอย่าง 8.13
การโอเวอร์โหลดดิ้งเมธอด

```python
class smartPhone:  # สร้างคลาสแม่
    def __init__(self):
        self.brand = 'iPhone'

    def openSmartPhone(self):
        return 'Access smartPhone with Passcode'

    def __str__(self):
        return f'{self.brand}: {self.openSmartPhone()}'


class smartPhoneOne(smartPhone):  # สร้างคลาสลูกที่ 1
    def __init__(self):
        super().__init__()

    def openSmartPhone(self):
        return 'Access smartPhone with Passcode or FingerID'


class smartPhoneTwo(smartPhoneOne):  # สร้างคลาสลูกที่ 2 สืบทอดคุณสมบัติมา
    ↪ จากคลาสลูกที่ 1
```

```
21      def __init__(self):
22          super().__init__()
23
24      def openSmartPhone(self):
25          return 'Access smartPhone with Passcode or FaceID'
26
27
28  a = smartPhone()  # สร้างอินสแตนซ์คลาสแม่
29  b = smartPhoneOne()  # สร้างอินสแตนซ์คลาสลูกที่ 1
30  c = smartPhoneTwo()  # สร้างอินสแตนซ์คลาสลูกที่ 2
31  b.brand = 'iPhone 5'  # กำหนดค่าแอตทริบิวต์แสดงผลคลาสลูกที่ 1
32  c.brand = 'iPhone X'  # กำหนดค่าแอตทริบิวต์แสดงผลคลาสลูกที่ 2
33  print(a)
34  print(b)
35  print(c)
```

```
iPhone: Access smartPhone with Passcode
iPhone 5: Access smartPhone with Passcode or FingerID
iPhone X: Access smartPhone with Passcode or FaceID
```

การทำโอเวอร์โหลดดิ้งเมธอดนี้ทำให้เราสามารถเปลี่ยนพฤติกรรม (เมธอด) ของคลาสลูกที่สืบทอดมาจากคลาสแม่ได้ เราจะสังเกตเห็นได้จากชื่อเมธอดคลาสแม่และคลาสลูกทั้งสองคลาาสที่สืบทอดต่อกันมาชื่อเมธอดเดียวกัน แต่มีข้อมูลที่แสดงผลแตกต่างกัน นอกจากนี้เราสามารถเปลี่ยนแปลงข้อมูลแอตทริบิวต์ที่อยู่ในคลาสแม่ให้แสดงผลที่คลาสลูกต่างจากคลาสแม่ได้ เห็นได้จากบรรทัดที่ 31-32 มีการกำหนดข้อมูลให้กับแอตทริบิวต์ `brand`

8.8 การห่อหุ้ม/ซ่อนข้อมูล

การเขียนโปรแกรมแบบ OOP เราสามารถกำหนดให้แอตทริบิวต์หรือเมธอดใด ๆ ในคลาสสามารถถูกเรียกใช้งานหรือเปลี่ยนแปลงแก้ไขข้อมูลจากการอ้างอิงผ่านอินสแตนซ์ได้ ซึ่งเป็นการป้องกันการเข้าถึงแอตทริบิวต์หรือเมธอดที่มีการกำหนดค่าไว้เรียบร้อยแล้ว ในบางครั้งเราอาจจะไปเปลี่ยนแปลงค่าข้อมูลตัวแปรโดยไม่ได้ตั้งใจ การห่อหุ้ม/หรือการซ่อนข้อมูล

(Data Encapsulation) จะทำให้ค่าข้อมูลที่อยู่ภายในคลาสมีความปลอดภัยมากขึ้น เมื่อต้องการป้องกันไม่ให้เข้าถึงแอตทริบิวต์หรือเมธอดใด ๆ ภายในคลาส ให้ใช้เครื่องหมาย __ (Double Underscore) นำหน้าชื่อแอตทริบิวต์หรือเมธอดนั้น ๆ แสดงดังตัวอย่างต่อไปนี้

ตัวอย่าง 8.14
การห่อหุ้มข้อมูลและเมธอดภายในคลาส

```
class Student:  # ประกาศสร้างคลาส Book
    def __init__(self):  # สร้างคอนสตรัคเตอร์และกำหนดแอตทริบิวต์
        self.ID = 685121
        self.fname = 'Tony'
        self.lname = 'Sinatra'
        self.__score = 65  # แอตทริบิวท์ที่ถูกป้องกันการเข้าถึง

    def ShowData(self):     # เมธอดแสดงข้อมูล
        print(self.ID, self.fname, self.lname, self.__score)

    def __updateData(self, degree):  # เมธอดที่ถูกป้องกันการเข้าถึง
        self.degree = degree
        print(self.ID, self.fname, self.lname, self.__score,
        ↪  self.degree)


std = Student()
std.ShowData()
std.__updateData('Master')
```

```
685121 Tony Sinatra 65
Traceback (most recent call last):
  File "/Users/epsilonxe/untitled.py", line 18, in <module>
    std.__updateData('Master')
AttributeError: 'Student' object has no attribute '__updateData'
```

ผลลัพธ์ที่ได้จากการสั่งให้โปรแกรมทำงานจะเห็นว่าเกิดความผิดพลาดขึ้น เพราะคลาส `Student` ไม่รู้จักแอตทริบิวต์ `__updateData` จากเรียกใช้งานผ่านอินสแตนซ์ที่สร้างขึ้น

มา ทำให้แสดงผลลัพธ์จากเมธอด `Showdata()` เท่านั้น สำหรับแอตทริบิวต์ `__score` ก็มีการป้องกันการเข้าถึงเช่นกัน หากเราทำการแก้ไขข้อมูลโดยตรงจะทำให้เกิดข้อผิดพลาดเช่นกัน สำหรับวิธีการนำข้อมูลจากเมธอด `__updateData` มาแสดงทำได้ ดังตัวอย่างต่อไปนี้

ตัวอย่าง 8.15
การเข้าถึงแอตทริบิวต์หรือเมธอดที่ถูกห่อหุ้ม

```
class Student:  # ประกาศสร้างคลาส Book
    def __init__(self):  # สร้างคอนสตรัคเตอร์และกำหนดแอตทริบิวต์
        self.ID = 685121
        self.fname = 'Tony'
        self.lname = 'Sinatra'
        self.__score = 65  # แอตทริบิวท์ที่ถูกป้องกันการเข้าถึง

    def ShowData(self):  # เมธอดแสดงข้อมูล
        print(self.ID, self.fname, self.lname, self.__score)

    def __updateData(self, degree):  # เมธอดที่ถูกป้องกันการเข้าถึง
        self.degree = degree
        print(self.ID, self.fname, self.lname, self.__score,
        ↪  self.degree)


std = Student()
std.ShowData()
std.__score = 50  # แก้ไขข้อมูลแอตทริบิวต์ __score
std._Student__updateData('Master')  # ข้อมูล __score มีค่าเท่าเดิม
std._Student__score = 50  # แก้ไขข้อมูลแอตทริบิวต์ __score ผ่านคลาส
↪  Student
std._Student__updateData('Master')  # ข้อมูลแอตทริบิวต์ __score เป็น 50
```

```
685121 Tony Sinatra 65
685121 Tony Sinatra 65 Master
685121 Tony Sinatra 50 Master
```

จากตัวอย่าง 8.15 เมื่อเราต้องการแก้ไขข้อมูลแอตทริบิวต์หรือต้องการเรียกใช้งานเมธอดที่ถูกป้องกันข้อมูลไว้ ให้เราใช้เครื่องหมาย _ (Underscore) ตามด้วยชื่อคลาสที่ต้องการเรียกใช้งานตามด้วยแอตทริบิวต์หรือเมธอด แสดงตามตัวอย่างในบรรทัดที่ 19-21 และจะสังเกตเห็นว่าในบรรทัดที่ 18 ถึงแม้ว่าเราทำการเปลี่ยนข้อมูลแอตทริบิวต์ `__score` แต่เมื่อแสดงผลออกมาค่าแอตทริบิวต์ของ `__score` ยังคงเป็น `65` เช่นเดิม

สรุปท้ายบท

ในบทนี้ได้นำเสนอวิธีการเขียนโปรแกรมแบบ OOP ซึ่งเป็นวิธีการเขียนโปรแกรมที่มีความซับซ้อนมากขึ้น แต่มีความยืดหยุ่นกว่าการเขียนโปรแกรมรูปแบบอื่น ๆ การเขียนโปรแกรมแบบ OOP มีวิธีการเขียนและเทคนิคอื่น ๆ อีกจำนวนมาก แต่ผู้อ่านก็ได้รู้จักวิธีการเบื้องต้น เช่น การสร้างคลาส เมธอด แอตทริบิวต์ การโอเวอร์โหลดดิ้งตัวดำเนินการ เป็นต้น การเขียนโปรแกรมรูปแบบนี้ค่อนข้างทำความเข้าได้ยากสำหรับผู้เริ่มต้น แต่ถ้าหากผู้เริ่มต้นฝึกฝนอยู่บ่อย ๆ การเขียนโปรแกรมแบบนี้จะกลายเป็นเรื่องที่ง่าย และโปรแกรมที่ถูกพัฒนาแบบ OOP นั้นจะสามารถแก้ไขและปรับปรุงโปรแกรมได้ง่าย

แบบฝึกหัด

1. จงสร้างคลาส `Vehicle` ที่มีแอตทริบิวต์ `name`, `max_speed` และ `mileage` และมีเมธอด `showProperty()` เพื่อแสดงข้อมูลของอินสแตนซ์ เช่น

   ```
   Vehicle -> Name: BMW i8, MaxSpeed: 225 kmph,
   ↪  Mileage: 12334 km
   ```

2. จงสร้างคลาส `Bus` ที่สืบทอดแอตทริบิวต์และเมธอดทั้งหมดจากคลาส `Vehicle`

3. จงสร้างคลาส `Bus` ที่สืบทอดจากคลาส `Vehicle` และ

   (a) มีแอตทริบิวต์ `max_capacity`

(b) โอเวอร์โหลดเมธอด `showProperty()` เพื่อแสดงข้อมูลของอินสแตนซ์ เช่น

```
Bus -> Name: Fuso Aero, MaxSpeed: 155 kmph,
↪  Mileage: 162334 km, MaxCapacity: 35
```

(c) มีเมธอด `showMaxCapacity()` ที่ใช้แสดงข้อมูลแอตทริบิวต์ `max_capacity` เช่น

```
The max capacity of Fuso Aero is 35
↪  passengers.
```