# 01076105, 01076106

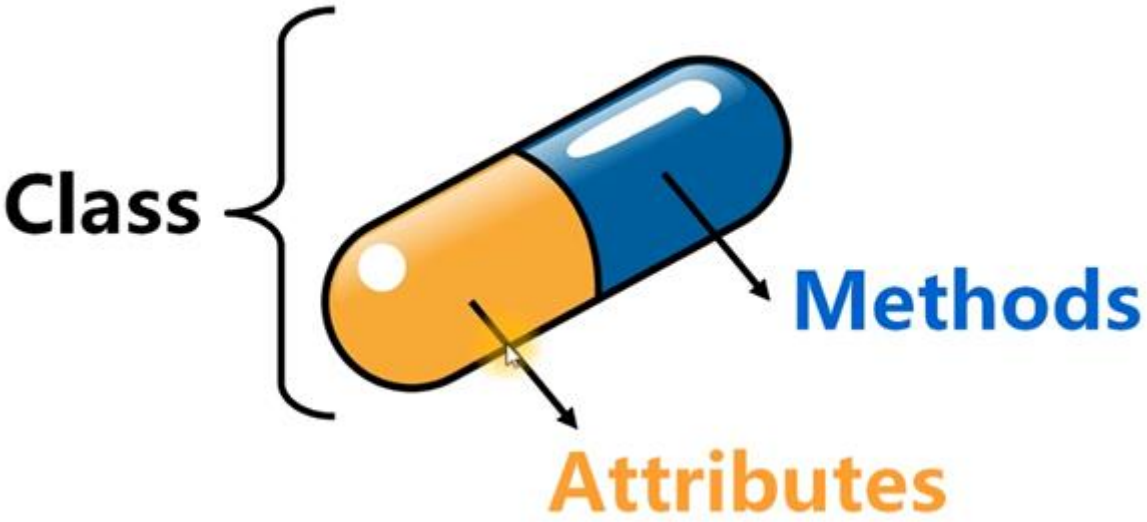
# Object Oriented Programming

# Object Oriented Programming Project

## Encapsulation, Method

# Encapsulation

- Encapsulation เป็นคุณสมบัติที่สำคัญหนึ่งของ Object Oriented โดย Encapsulation มาจากคำว่า capsule: (เป็นชิ้นเดียวกัน) กับ en = (ทำให้)
- ความหมาย คือ การนำข้อมูล (attribute) กับการทำงาน (method) มารวมไว้ด้วยกัน และป้องกันไม่ให้ภายนอกสามารถเข้าถึงข้อมูลที่ไม่ต้องการได้



# Encapsulation

- ลักษณะอย่างหนึ่งที่เกิดขึ้นจาก Encapsulation คือ information hiding ซึ่งหมายถึง การซ่อนหรือจำกัดการเข้าถึง “ข้อมูล” บางส่วนที่ไม่ต้องการให้เข้าถึงจากภายนอก object ได้โดยตรง
- ยกตัวอย่าง เช่น ใน object Student หากปล่อยให้มีการแก้ไข ชื่อนักศึกษาจากที่ไหนก็ได้ เท่ากับว่า ไม่มีการปกป้องข้อมูลของ object อย่างเพียงพอ

```
<object>.<attribute> = <new_value>
```

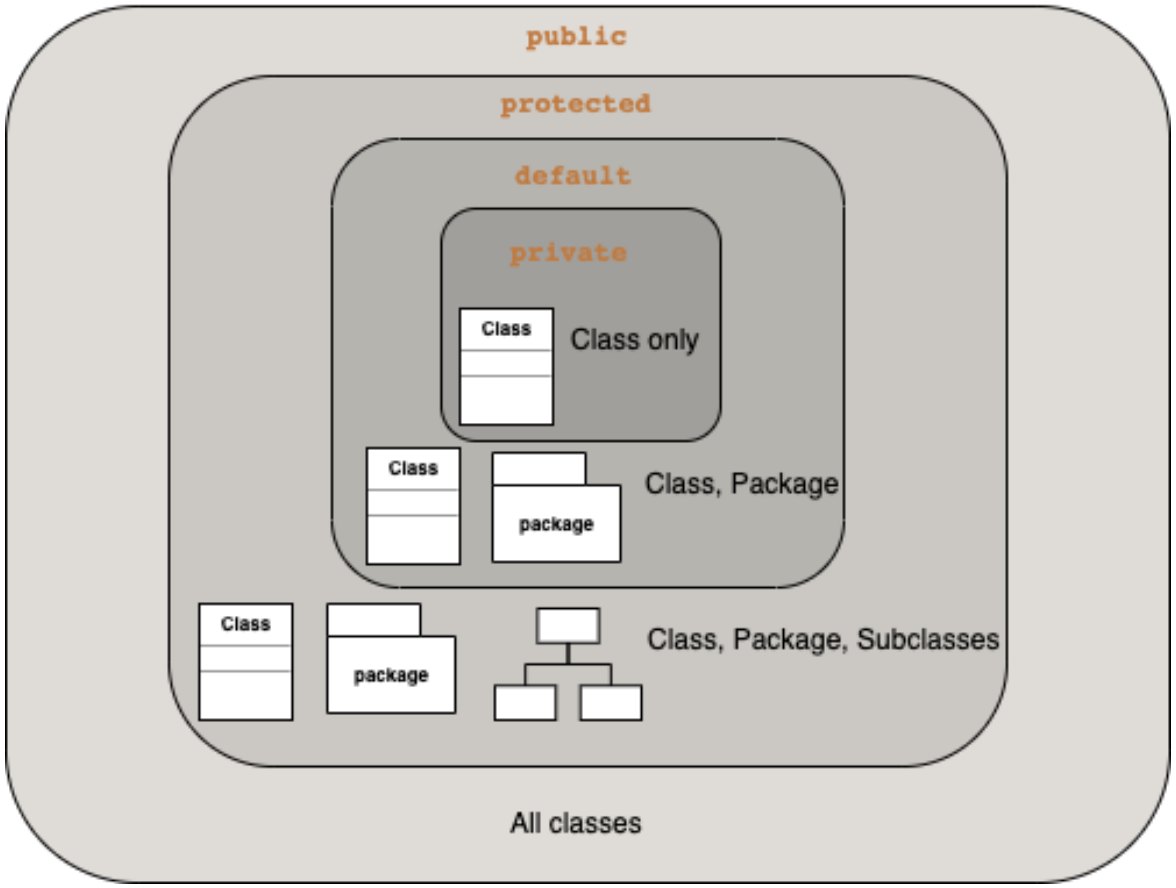
- การเข้าถึง attribute ของ Instance โดยใช้ dot notation จึงขัดกับหลัก information hiding จึงต้องจำกัดการเข้าถึงข้อมูล

# Encapsulation

- ในภาษา Programming ที่เป็น Object Oriented โดยทั่วไปจะแบ่งประเภทข้อมูลของ Object เอาไว้ 2-3 ระดับ ในหลายภาษามีการแบ่งดังนี้
  - ข้อมูลระดับที่ 1 คือ Public หมายถึงข้อมูลที่อนุญาตให้เข้าถึงจากภายนอก Object ได้โดยตรงโดยอิสระ
  - ข้อมูลระดับที่ 2 คือ Protected หมายถึงข้อมูลที่อนุญาตให้เข้าถึง จากภายในคลาสของตนเอง และคลาสที่สืบทอดไปเท่านั้น
  - ข้อมูลระดับที่ 3 คือ Private หมายถึงข้อมูลที่อนุญาตให้เข้าถึง จากภายในคลาสของตนเองเท่านั้น

# Encapsulation

- การจำกัดการเข้าถึง เรียกว่า access modifier จากรูปเป็น access modifier ของภาษา JAVA



# Encapsulation

- ภาษา Java จะเข้มงวดกับ access modifier มาก โดยภาษาเองจะทำหน้าที่ป้องกันการเข้าถึงตามที่กล่าวมาก
- แต่ภาษา Python ไม่ได้ใช่วิธี access modifier ในการควบคุมการเข้าถึง แต่ใช้วิธี Name Conventional โดยกำหนดให้แนวทางการตั้งชื่อให้เป็นไปตามตารางนี้
- Python ไม่ได้บังคับการเข้าถึง แต่ใช้แนวทางการตั้งชื่อเพื่อให้ทราบว่าต้องการแบบใด

| Name | Notation | Behavior |
|---|---|---|
| name | Public | เข้าถึงได้จากภายในและภายนอก |
| _name | Protected | เหมือนกับ Public แต่ไม่ควรเข้าถึงจากภายนอก |
| __name | Private | ไม่สามารถเห็นได้จากภายนอก |

# Encapsulation

- การใช้ก่อนหน้านี้เป็น public สำหรับการใช้แบบ protected ตามตัวอย่างด้านล่าง
- จากโปรแกรมนี้ จะเห็นว่ายังสามารถอ้างถึง _year ได้ แต่อ้าง year เฉยๆ ไม่ได้
- ดังนั้นการตั้งชื่อ attribute ในภาษา python ให้เป็น protected ให้ใช้ _นำหน้า

main.py

```python
class Car:
    def __init__(self, brand, model, year):
        self._brand = brand
        self._model = model
        self._year = year

my_car = Car("Porsche", "911 Carrera", 2022)

print(my_car._year)
my_car._year = 2024
print(my_car._year)

#print(my_car.year)
```

```
2022
2024
```

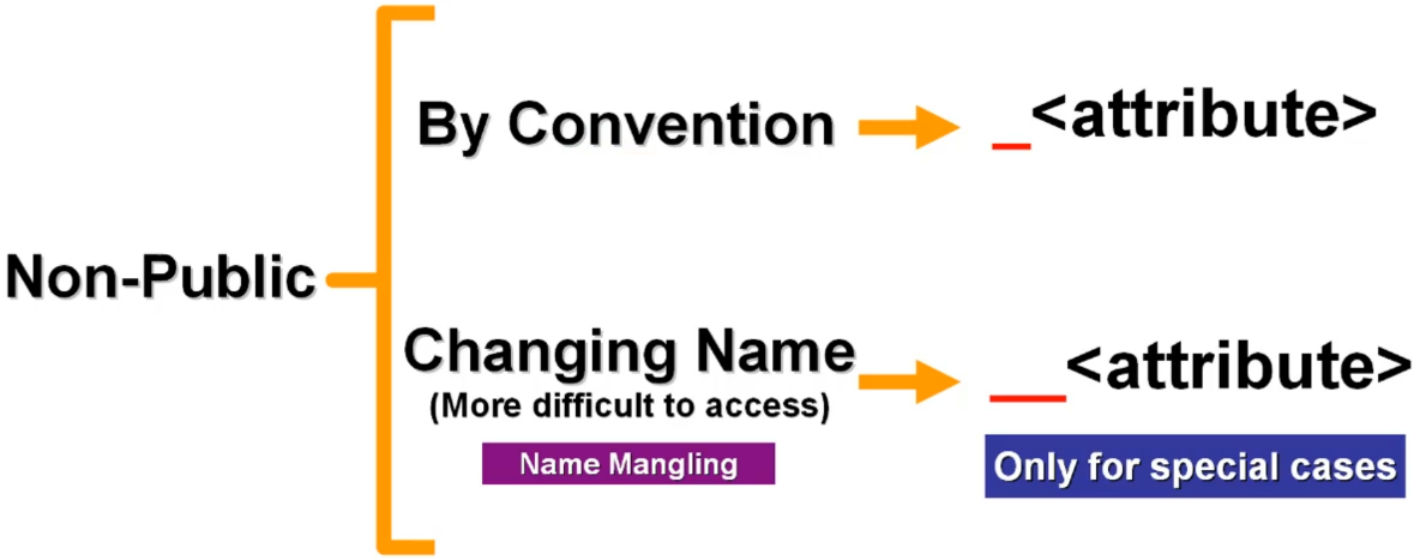
# Encapsulation

- สำหรับการกำหนดข้อมูลเป็น private โดยใช้เครื่องหมาย __ (double under) นั้น แม้จะใช้การอ้างถึงโดยตรง ก็ยังไม่สามารถทำได้ เพราะถูกป้องกันโดย Python

```
class Car:

    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.__year = year

my_car = Car("Porsche", "911 Carrera", 2020)

#print(my_car.year) # Can't be accessed
#print(my_car._year) # Can't be accessed
print(my_car.__year)
```

# Encapsulation

- ในบางตำราจะไม่แยก attribute ของคลาสใน Python เป็น public, protected และ private แต่จะเรียกเป็น Public กับ Non-Public
- จากนั้นจึงค่อยแยก Non-Public ออกเป็น 2 ประเภท ตามรูป
- และไม่ควรอ้างตัวแปรในคลาสโดยตรง โดยใช้ “_”



# Setter and Getter

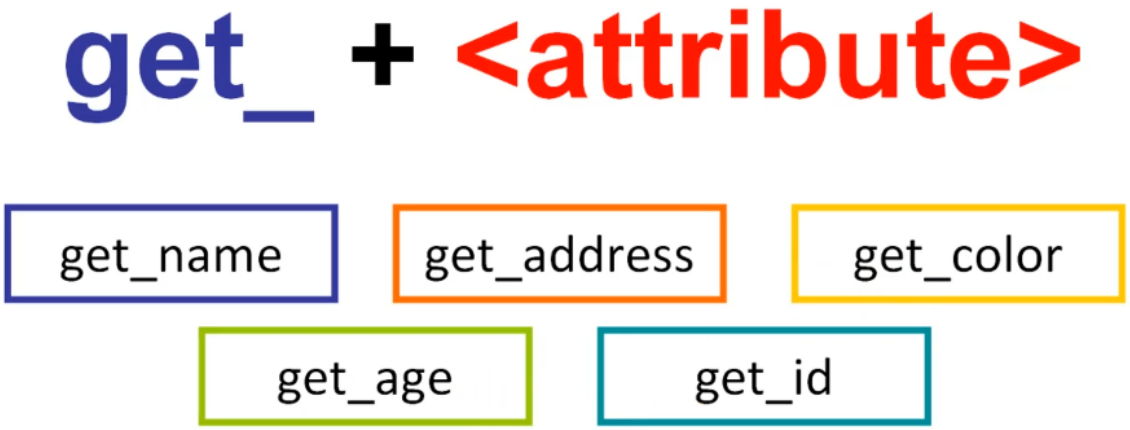
- จากหลักการ encapsulation และ การจำกัดการเข้าถึงโดยใช้ access modifier ดังนั้นการจะเข้าถึง attribute ของคลาสโดยตรง จะทำไม่ได้ แล้วจะทำอย่างไร?
- วิธีการที่ใช้ คือ ให้เข้าถึงผ่านฟังก์ชัน ซึ่งฟังก์ชันในคลาสจะมีชื่อเรียกว่า method
  - จะเรียก method ที่ทำหน้าที่ อ่านข้อมูล attribute ว่า getters
  - จะเรียก method ที่ทำหน้าที่ เปลี่ยนแปลงข้อมูล attribute ว่า setter

**Getters** → **Get** the value of an attribute.

**Setters** → **Set** the value of an attribute.

# Setter and Getter

- getter เป็น method สำหรับอ่านค่าจาก attribute มักใช้คำว่า get + “_” จากนั้นตามด้วยชื่อ attribute ตามตัวอย่างในรูป
- ไม่จำเป็นว่าทุก attribute จะต้องมี getter ถ้า attribute ใดที่ต้องการให้อ่านค่าจากภายนอกได้ ให้ทำ getter ไว้ แต่ถ้า attribute ใด ใช้เฉพาะในคลาส ก็ไม่ต้องทำ



# Setter and Getter

- setter เป็น method สำหรับกำหนดค่าให้กับ attribute ใน instance มักใช้คำว่า set และ “_” จากนั้นตามด้วยชื่อ attribute
- setter มีหน้าที่สำคัญ เพราะต้องทำหน้าที่ validate ข้อมูล เมื่อข้อมูลอยู่ในช่วงที่ถูกต้อง จึงจะกำหนดค่าได้ ทำให้การควบคุมค่าของข้อมูลทำได้มากขึ้น

**set_ + <attribute>**

# Setter and Getter

- จากตัวอย่างคลาส Student ในครั้งก่อน หากจะปรับปรุงให้มีคุณสมบัติเรื่อง encapsulation โดยการเปลี่ยน access modifier และเพิ่ม getter และ setter เข้าไป ก็จะได้คลาส Student ดังนี้
- จะเห็นว่า การกำหนดค่าให้ attribute จะต้องกระทำผ่าน setter เท่านั้น

```python
class Student:
    def __init__(self, stu_id, stu_name):
        self.__stu_id = stu_id
        self.__stu_name = stu_name

    def get_id(self):
        return self.__stu_id

    def set_id(self, stu_id):
        if stu_id.isnumeric() and len(stu_id) == 8:
            self.__stu_id = stu_id
        else:
            raise ValueError("Invalid ID")

stu1 = Student('001' ,'John')
stu2 = Student('002', 'Peter')
print(stu1.get_id())
stu1.set_id('66010100')
stu1.set_id('6601020')
```

# Setter and Getter

- Activity #1 จาก Activity #2 ของครั้งที่แล้ว
  - ให้เขียนโปรแกรม เพื่อสร้างคลาสต่อไปนี้
    - นักศึกษา (Student) โดยมี attribute : stu_id, stu_name
    - รายวิชา (Subject) โดยมี attribute : subject_id, subject_name, section, credit
    - ผู้สอน (Teacher) โดยมี attribute : teacher_id, teacher_name
- ให้ปรับปรุง Code โดยเพิ่ม access modifier กำหนดให้ attribute ทุกตัวในคลาสเป็นแบบ private
- ให้เขียน setter และ getter ของทุก attribute ที่จำเป็น โดยให้ validate ข้อมูลในส่วน setter ด้วย เช่น ชื่อ ต้องเป็นภาษาอังกฤษเท่านั้น

# Methods

- นอกเหนือจาก getter และ setter แล้ว ในคลาสยังต้องมีฟังก์ชันหรือ method ที่เป็นภาระงานของคลาสนั้นด้วย โดย method จะต่างจากฟังก์ชัน คือ สามารถเข้าถึง attribute (state) ของเฉพาะ instance นั้น
- ในการเขียน methods จะต้องมีคำว่า self เพื่อใช้ในการอ้างอิงถึง instance ที่เรียกใช้ method แม้จะไม่มี พารามิเตอร์ เลยก็ตาม

```
class MyClass:

    # Class Attributes

    # __init__()

    def method_name(self, param1, param2, ...):
        # Code
```

# Methods

- ชื่อของ method ควรเป็นคำกริยา เพื่อแสดงว่า method นี้ “ทำ” อะไร
- ควรใช้ snake case (อักษรตัวเล็ก คั่นด้วย _ ) เพื่อให้อ่านง่าย
- ถ้าเป็น protected method ควรขึ้นต้นด้วย _


- Build
- Show
- Shuffle
- Draw Card
- More…

# Methods

- ตัวอย่างเช่น ในคลาส Subject ซึ่งจะต้องเก็บข้อมูลของนักศึกษาที่ลงเรียนในวิชานั้น จะต้องมี method สำหรับการเพิ่มชื่อนักศึกษาลงใน List ของตัวเอง

```
class Subject:
    def __init__(self, subject_id, subject_name):
        self.sub_id = subject_id
        self.sub_name = subject_name
        self.student_list = []

    def add_student(self, student):
        if isinstance(student, Student):
            self.student_list.append(student)
```

- method ตัวอย่างนี้ method ไม่ส่งค่ากลับ แต่บาง method อาจส่งค่าออกไปนอกคลาสก็ได้

# Methods

- การเรียกใช้ Method จะคล้ายกับเรียก function แต่ระบุชื่อ instance ด้วย


```
student1 = Student('66010001' ,'John')
student2 = Student('66010002', 'Peter')
print(student1.get_id())
student2.set_id('66010100')
subject1 = Subject('01076140', 'Calculus')
subject1.add_student (student2)
```

# Methods

- ใน Class แต่ละ method สามารถเรียกใช้ระหว่างกันได้ ตามตัวอย่าง

```
class Subject:
    def __init__(self, subject_id, subject_name):
        self.sub_id = subject_id
        self.sub_name = subject_name
        self.student_list = []

    def add_student(self, student):
        if isinstance(student, Student):
            self.student_list.append(student)

    def add_student_list(self, student_list):
        for st in student_list:
            self.add_student(st)
```

# Methods : chaining

- ลองดู Class ต่อไปนี้ (ดูที่ add_topping) สามารถจะส่งคืน instance เองได้

```
1  class Pizza:
2
3      def __init__(self):
4          self.toppings = []
5
6      def add_topping(self, topping):
7          self.toppings.append(topping.lower())
8          return self
9
10     def display_toppings(self):
11         print("This Pizza has:")
12         for topping in self.toppings:
13             print(topping.capitalize())
```

# Methods : chaining

- จะเห็นคำสั่ง return self ซึ่งเป็นการ return instance ที่เรียกใช้ method
- ทำให้เราสามารถทำ method chaining ได้ ตามตัวอย่าง

```
1  pizza.add_topping("mushrooms") \
2      .add_topping("olives") \
3      .add_topping("chicken") \
4      .display_toppings()
```

# Method __str__

- ใน Python จะมี method พิเศษ ที่ขึ้นต้นและปิดท้ายด้วย __ เรียกว่า dunder (ย่อมาจาก double under) จำนวนหนึ่ง ซึ่งจะกล่าวถึงโดยละเอียดภายหลัง
- method ที่น่าสนใจ คือ __str__ ซึ่งจะเป็น method ที่จะถูกเรียกใช้เมื่อ print object

main.py

```
1  class MyClass:
2
3      def __init__(self, anyNumber, anyString):
4          self.x = anyNumber
5          self.y = anyString
6
7      def __str__(self):
8          return 'MyClass(x=' + str(self.x) + ' ,y=' + self.y + ')'
9
10 myObject = MyClass(12345, "Hello")
11
12 print(myObject.__str__())
13 print(myObject)
14 print(str(myObject))
```

```
MyClass(x=12345 ,y=Hello)
MyClass(x=12345 ,y=Hello)
MyClass(x=12345 ,y=Hello)
```

# Property Class

- การใช้ getter และ setter สร้าง information hiding ตามแนวคิด encapsulation
- แต่ข้อเสียคือ แทนที่จะให้ความรู้สึกของการเข้าถึง attribute แบบเดิม กลับต้องทำผ่าน method ซึ่งทำให้โปรแกรมดูยุ่งยาก ไม่เหมือนกับการเข้าถึง attribute
- อย่างไรก็ตาม Python ได้ให้คลาส Property ไว้ เพื่อทำให้การเรียก getter และ setter เป็นไปโดยสะดวกมากขึ้น

```
<property_name> = property(<getter>, <setter>)
```

# Property Function

- Property เป็นคลาสของ Python ที่ช่วยให้ใช้งานคล้ายกับการไม่ใช้ setter/getter
- จากรูป age จะเป็น instance ของคลาส Property โดยมีฟังก์ชัน get_age, set_age เป็น argument
- เมื่อมีการเรียก my_dog.age ถ้าเป็นการอ่านค่า Python จะเรียกฟังก์ชัน get_age มาทำงาน
- แต่หากมีการเปลี่ยนแปลงค่าใน my_dog.age จะเรียกฟังก์ชัน set_age มาทำงาน ทำให้คล้ายกับการเข้าถึง attribute โดยตรง

```python
class Dog:

    def __init__(self, age):
        self.__age = age

    def get_age(self):
        return self.__age

    def set_age(self, new_age):
        if isinstance(new_age, int) and 0 < new_age < 30:
            self.__age = new_age
        else:
            print("Please enter a valid age.")

    age = property(get_age, set_age)

my_dog = Dog(8)
print(f"My dog is {my_dog.age} years old.")
print("One year later...")
my_dog.age += 1
print(f"My dog is now {my_dog.age} years old.")
```

# Property Function

- จะเห็นว่าการใช้งาน สามารถอ้างถึง attribute **age** ได้คล้ายกับไม่ได้ใช้ getter และ setter
- แต่มีข้อดีมากกว่า เพราะสามารถ **validate** ข้อมูลได้ (กรณี setter)
- แต่มีปัญหาเกิดขึ้นเล็กน้อย เพราะเท่ากับว่าสามารถอ้างถึง attribute ได้ ถึง 2 วิธี คือ ใช้ **set_age(8)** ก็ได้ หรือ **my_dog.age = 8** ก็ได้ เพราะฟังก์ชัน setter เดิมก็ยังอยู่ และ ใช้ object age ที่เกิดจาก property ก็ได้

```
my_dog.age += 1
print(f"My dog is now {my_dog.age} years old.")
my_dog.set_age(my_dog.get_age()+1)
print(f"My dog is now {my_dog.age} years old.")
```

# Closures

- Closures เป็นอีกคุณสมบัติที่มีในภาษา Programming สมัยใหม่ เช่น Python หรือ Javascript โดยเป็นคุณสมบัติที่ต่อยอดมาจาก first class function
- จากรูปจะเห็นว่าเมื่อเรียก outer_func() จะมี return ค่า inner_func() ซึ่งจะเห็นว่า inner_func จะยังสามารถเข้าถึงตัวแปร x ได้ (กรณีนี้เรียกว่า free variable เพราะไม่อยู่ภายใน inner_func()) จึงเรียกคุณสมบัตินี้ว่า closures

```
def outer_func():
    x = 6
    def inner_func():
        print("Value of x from inner::",x)

    return inner_func

out = outer_func()
out()
```

```
Value of x from inner:: 6
```

# Closures

- เอาความสามารถนี้ไปทำอะไรได้บ้าง ลองดูตัวอย่าง จะเห็นว่าเราสามารถสร้าง function ที่ทำงานต่างกันเล็กน้อยได้ จาก source code ชุดเดียวกัน

```
def outer_func(a):

    def inner_func():
        print("Value of a from inner::",a)
    return inner_func

inner = outer_func(90)
inner()
inner2 = outer_func(200)
inner2()
```

```
Value of a from inner:: 90
Value of a from inner:: 200
```

# Closures

- ลองดูอีกตัวอย่าง คราวนี้จะให้ inner_func รับพารามิเตอร์ด้วย จะเห็นว่าฟังก์ชัน inner_func สามารถจะเข้าถึงพารามิเตอร์ a ซึ่งเป็น free_variable และ b ที่ส่งผ่านพารามิเตอร์ภายหลัง

```
def outer_func(a):

    def inner_func(b):
        print("Value of a from inner::",a)
        print("Value of b passed to inner::",b)

    return inner_func

inner = outer_func(90)
inner(200)
```

```
Value of a from inner:: 90
Value of b passed to inner:: 200
```

# Closures

- สรุปเงื่อนไขในการใช้งาน closures
  - เมื่อฟังก์ชันมีการซ้อนกัน (Nested)
  - ฟังก์ชันด้านในมีการอ้างถึงตัวแปรที่อยู่ที่ฟังก์ชันด้านนอก
  - มีการ return ฟังก์ชันด้านในจากฟังก์ชันด้านนอก

# Decorator

- ลองดูตัวอย่างต่อไปนี้ เมื่อทำงานจะแสดงผลอย่างไร
- จะเห็นว่าฟังก์ชัน make_pretty จะรับพารามิเตอร์เป็นฟังก์ชันใดๆ และเพิ่มการทำงานจากฟังก์ชันนั้นเข้าไปอีก จึงเรียกการทำงานแบบนี้ว่า decorator (ตกแต่ง)

```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def ordinary():
    print("I am ordinary")

decorated_func = make_pretty(ordinary)

decorated_func()
```

```
I got decorated
I am ordinary
```

# Decorator

- ในภาษา Python สามารถใช้ decorator ในรูปแบบของ shortcut ได้ โดยใช้เครื่องหมาย @ ดังนั้น @make_pretty จึงมีความหมายว่าให้นำฟังก์ชัน ordinary ไปตกแต่งด้วยฟังก์ชัน make_pretty และสามารถใช้งานในชื่อ ordinary เหมือนเดิม

```
def make_pretty(func):

    def inner():
        print("I got decorated")
        func()
    return inner

@make_pretty
def ordinary():
    print("I am ordinary")

ordinary()
```

# @property Decorator

- ซึ่ง property ของ Python ก็สามารถใช้แบบ decorator ได้ ตามตัวอย่าง ซึ่งจะทำให้สามารถใช้งาน my_dog.age ได้ และสามารถเข้าถึงได้ทางเดียว (set_age ใช้ไม่ได้)

```
class Dog:
    def __init__(self, age):
        self.__age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, new_age):
        if isinstance(new_age, int) and 0 < new_age < 30:
            self.__age = new_age
        else:
            print("Please enter a valid age.")
```

# @property Decorator

- รูปแบบการใช้งาน getter จะเขียนดังนี้

```
@property
def property_name(self):
    return self._property_name
```

- และรูปแบบการใช้งาน setter จะเขียนดังนี้

```
@property_name.setter
def property_name(self, new_value):
    self._property_name = new_value
```

- ให้พิจารณาว่า attribute ใดจำเป็นต้องมี getter หรือ setter บ้าง


# For your attention