



01076105, 01076106

Object Oriented Programming

Object Oriented Programming Project

Method, Use Case Diagram



Setter and Getter

- จากหลักการ Encapsulation และ การจำกัดการเข้าถึงโดยใช้ Access Modifier ดังนั้นการจะเข้าถึง attribute ของคลาส จึงต้องกระทำผ่าน method เท่านั้น
- หลักนิยมของ OOP
 - จะเรียก method ที่ทำหน้าที่ อ่านข้อมูล ว่า getters
 - จะเรียก method ที่ทำหน้าที่ เปลี่ยนแปลงข้อมูลว่า 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 จะต้องมีการมี getter ถ้า attribute ใดที่ต้องการให้อ่านค่าจากภายนอกได้ ให้ทำ getter ไว้ แต่ถ้า attribute ใด ใช้เฉพาะในคลาส ก็ไม่ต้องทำ

get_ + <attribute>

get_name

get_address

get_color

get_age

get_id



Setter and Getter

- จะเห็นว่าไม่สามารถเข้าถึง `__title` ได้จากภายนอก class จะต้องกระทำผ่าน getter เท่านั้น

```
class Movie:

    def __init__(self, title, rating):
        self.__title = title
        self.__rating = rating

    def get_title(self):
        return self.__title

my_movie = Movie("The Godfather", 4.8)

print(my_movie.title) # Throws an error

print(my_movie.get_title())
print("My favorite movie is:", my_movie.get_title())
```



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>

set_name

set_address

set_color

set_age

set_id



Setter and Getter

- จะเห็นว่า การกำหนดค่าให้ attribute จะต้องกระทำผ่าน setter เท่านั้น
- ใน setter จะมีการตรวจสอบชนิดของข้อมูล และ ตรวจสอบว่าเป็นตัวอักษรอย่างเดียว

```
class Dog:

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

    def get_name(self):
        return self.__name

    def set_name(self, new_name):
        if isinstance(new_name, str) and new_name.isalpha():
            self.__name = new_name
        else:
            print("Please enter a valid name.")

my_dog = Dog("Nora", 8)
print("My dog is:", my_dog.get_name())
my_dog.set_name("Norita")
print("Her new name is:", my_dog.get_name())
```



Setter and Getter

- ตัวอย่าง set_items แรก จะผิดประเภทของข้อมูล จะทำไม่ได้

```
class Backpack:

    def __init__(self):
        self._items = []

    def get_items(self):
        return self._items

    def set_items(self, new_items):
        if isinstance(new_items, list):
            self._items = new_items
        else:
            print("Please enter a valid list of items.")

my_backpack = Backpack()
print(my_backpack.get_items())

my_backpack.set_items("Hello, World!") # Invalid value
my_backpack.set_items(["Water Bottle", "Sleeping Bag", "First Aid Kit"])

print(my_backpack.get_items())
```



Property Class

- การใช้ getter และ setter เพื่อให้เกิด information hiding ตามแนวคิด Encapsulation
- แต่ข้อเสียคือ แทนที่จะให้ความรู้สึกรู้สึกของการเข้าถึง attribute แบบเดิม กลับต้องทำผ่าน method ซึ่งทำให้โปรแกรมดูยุ่งยาก ไม่เหมือนกับการเข้าถึง attribute
- อย่างไรก็ตาม Python ได้ให้คลาส Property ไว้ เพื่อให้การเรียก 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

```
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

- คำสั่ง `dir` จะใช้ในการแสดง method ของ object จะเห็นว่ามี method `set` และ `get`

```
print(Dog.age.fget)
print(Dog.age.fset)
print(dir(Dog.age))
```

```
<function Dog.get_age at 0x7fdf6cd05b80>
<function Dog.set_age at 0x7fdf6cd05c10>
['__class__', '__delattr__', '__delete__', '__dir__', '__doc__', '
__eq__', '__format__', '__ge__', '__get__', '__getattr__', '
__gt__', '__hash__', '__init__', '__init_subclass__', '__isabstract
method__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__set__', '__setattr__', '__sizeof__
', '__str__', '__subclasshook__', 'deleter', 'fdel', 'fget', 'fset
', 'getter', 'setter']
>
```



Property Function

- จะเห็นว่าการใช้งาน สามารถอ้างถึง attribute **age** ได้คล้ายกับไม่ได้ใช้ getter และ setter
- แต่มีข้อดีมากกว่า เพราะสามารถ **validate** ข้อมูลได้ (กรณี setter)
- แต่มีปัญหาก่เกิดขึ้นเล็กน้อย 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)
 - ฟังก์ชันด้านในมีการอ้างถึงตัวแปรที่อยู่ฟังก์ชันด้านนอก
 - 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 ซึ่งจะช่วยให้สามารถใช้งาน 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

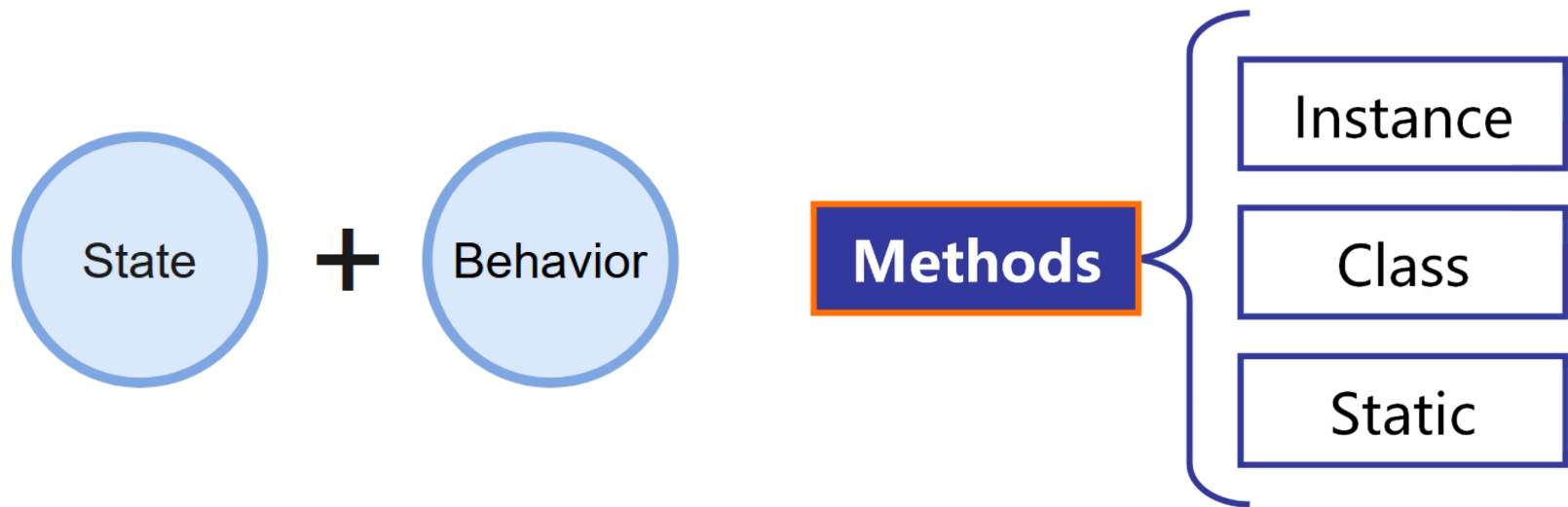
```
@property_name.setter
def property_name(self, new_value):
    self._property_name = new_value
```

- ให้พิจารณาว่า attribute ใดจำเป็นต้องมี getter หรือ setter บ้าง



Methods

- เนื่องจาก คลาส ประกอบด้วย attribute และ method โดย attribute ทำหน้าที่เก็บสถานะของ object และ method ทำหน้าที่กำหนด พฤติกรรมของ object
- Method จะแบ่งออกเป็น 3 ประเภท คือ instance method, class method และ static method





Methods

- Instance methods คือ methods ที่เป็นของ instance ใดๆ โดยสามารถเข้าถึง attribute (state) ของเฉพาะ instance นั้น
- ดังนั้น methods ประเภทนี้จึงต้องมีคำว่า **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

- ตัวอย่างของ method ตัวอย่างนี้ method จะไม่ส่งค่ากลับ

```
class Circle:

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

    # Printing the value
    def find_diameter(self):
        print(f"Diameter: {self.radius * 2}")
        # The value could be returned too with:
        # return self.radius * 2
```



Methods

- ตัวอย่างของ Method ที่มีการส่งคืนค่า

```
class Backpack:
    def __init__(self):
        self._items = []

    @property
    def items(self):
        return self._items

    def add_item(self, item):
        if isinstance(item, str):
            self._items.append(item)
        else:
            print("Please provide a valid item.")

    def remove_item(self, item):
        if item in self._items:
            self._items.remove(item)
            return 1
        else:
            return 0

    def has_item(self, item):
        return item in self._items
```




Methods

- การเรียกใช้ Method จะคล้ายกับเรียก function แต่ระบุชื่อ instance ด้วย


`<object>.<method>(<arguments>)`

```
my_list = [4, 5, 6, 7, 8]
```

```
my_list.sort()
```

```
print(my_list)
```

```
my_list.append(14)
```

```
print(my_list)
```

```
my_list.extend([1, 2, 3])
```

```
print(my_list)
```



Methods

- ใน Class แต่ละ method สามารถเรียกใช้ระหว่างกันได้ ตามตัวอย่าง

```
class Backpack:

    def __init__(self):
        self._items = []

    @property
    def items(self):
        return self._items

    def add_multiple_items(self, items):
        for item in items:
            self.add_item(item)

    def add_item(self, item):
        if isinstance(item, str):
            self._items.append(item)
        else:
            print("Please provide a valid item.")
```



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

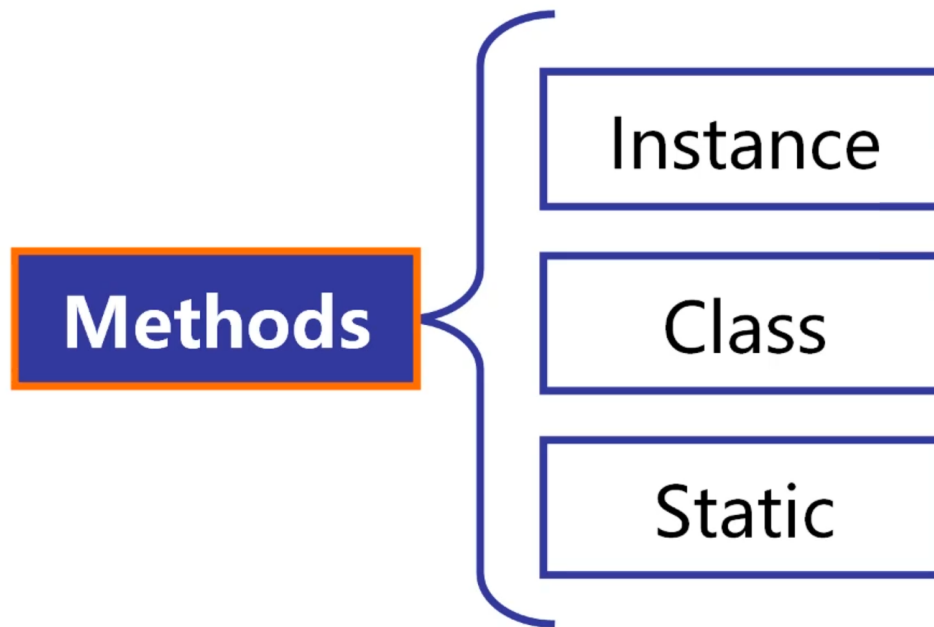
```
main.py
1 class MyClass:
2
3     def __init__(self, anyNumber, anyString):
4         self.x = anyNumber
5         self.y = anyString
6
7     def __str__(self):
8         return 'MyClass(x=' + str(self.x) + ' ,y=' + self.y + ' )'
9
10 myObject = MyClass(12345, "Hello")
11
12 print(myObject.__str__())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

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



Methods

- นอกเหนือจาก instance method แล้ว ยังมี method อีก 2 ประเภทได้แก่
 - Static method คือ method ที่ไม่มีการใช้ attribute ในคลาส ดังนั้นจึงสามารถเรียกใช้ได้โดยไม่ต้องสร้าง instance ก่อน
 - มักจะเป็นฟังก์ชันทั่วไป ที่เอาไปฝากไว้ที่คลาส เนื่องจากมีการทำงานที่ใกล้เคียงกัน





Static method

- เป็น method ที่สามารถเรียกใช้ได้โดยไม่ต้องสร้าง instant

```
class Student:
    def __init__(self, name, height):
        self._name = name
        self._weight = weight
        self._height = height

    @staticmethod
    def kg_to_pound(kg):
        return kg * 2.20462

    def cm_to_inch(cm):
        return cm * 0.393701

print(Student.kg_to_pound(50))
```



Class method

- เป็นคลาสที่ใช้ในการทำ Constructor แบบอื่นๆ ได้ (cls คือ constructor)

```
main.py
1 class Point:
2     def __init__(self, x, y):
3         self._x = x
4         self._y = y
5
6     @classmethod
7     def of(cls, point_string):
8         s = point_string.split("-")
9         return cls(int(s[0]),int(s[1]))
10
11 p1 = Point(5, 5)
12 p2 = Point.of("10-10")
13 print(p2._x)
```




For your attention