Python類別與物件簡介

馬誠佑

林口長庚紀念醫院/醫療人工智慧核心實驗室

- 類別(Class)
- 物件(Object)
- 屬性(Attribute)
- 建構式(Constructor)
- 方法(Method)
- 繼承
- 多型
- 特殊函數(方法)

Class vs. Object

- OOP (Object-Oriented Programming)
- Class : 物件的藍圖、自訂義的資料型態 (內含data與method)
- Object: 透過class建立的實體(instance)
- 判斷class與instance的關係 isinstance(object_name, class_name)

區分	傳統的開發 Traditional development	物件導向的開發 Object-Oriented Development
方法 Method	程序導向 Procedure-Oriented	物件導向 Object-Oriented
分解基礎 Decomposition based on	演算法 Algorithm	類別 Class
生命週期 Life Cycle	由上而下 Top-Down	往覆與漸增 Iterative and Incrementally
開發系統的可維護性 Maintainability of Developed System	困難	容易
可再使用性 Reusability	低	高
失敗與風險 Failure and Risk	高	低

https://www.dotblogs.com.tw/gelis/2011/03/17/21892

Class

```
class ClassName:
    def __init__(self, para1, para2, …):
        self.variable1 = …
        self.variable2 = …
        ….
        ….
    def classFunction1(self, para1, para2, …):
        ….
        ….
```

```python
class Car:
    def __init__(self, wheels_number=4, car_doors=4, passengers=4,
color='white',driveMode=''):
        self.wheels_number = wheels_number
        self.car_doors = car_doors
        self.passengers = passengers
        self.color = color
        self.driveMode = driveMode

    def drive(self):
        print('driving the car.')

toyota = Car(4, 5, 5)
toyota.drive()
bmw = Car(4, 2, 2)
bmw.drive()
```

Attribute

```
#object_name.attribute_name = value

benz = Car()
benz.color = 'silver'
benz.car_doors = 5
benz.driveMode = 'FF'
```

Constructor

- 於建立物件(Object)的同時，會自動執行的方法(Method)，用以初始化物件的屬性，至少要有一個self參數。

```
def __init__(self, wheels_number=4, car_doors=4, passengers=4,
color='white',driveMode=''):
        self.wheels_number = wheels_number
        self.car_doors = car_doors
        self.passengers = passengers
        self.color = color
        self.driveMode = driveMode
```

Practice

- 請寫一個student類別屬性有名子、性別、生日、成績，成績使用字典來儲存{‘科目’:‘分數’}的各科成績，撰寫__init__()函式，
  - 在產生學生A及學生B，同時輸入各科成績，如未輸入則預設為0分(也就是缺考)。
  - A的成績分別為：數學60、英文70、物理95
  - B的成績分別為：數學100、英文55、物理100

- 承上題，請新增一名Student，其名子為”顆顆”，數學成績90、英文成績60，物理成績為33。

- 請為Student新增一個方法，讓兩個Student可以互相比較，名稱為compare()。

例如A.compare(B)，假設：

A的總分高於B -> A的名字 + '贏了！'

A的總分等於B -> '什麼？竟然平手？！'

A的總分小於B -> '可...可惡，難道，這就是' + B的名字 + '真正的實力嗎？'

Class Attribute vs. Instance Attribute

```
class Cat:
    color = 'gray'
    def __init__(self,name=''):
        self.name = name
    def printColor(self):
        print(self.color)

milk = Cat('!!milk!!')
print(milk.name)

milk.printColor()
milk.color = 'yellow'
milk.printColor()
#cheese.printColor()
#kk = cat()
print('class attr of Cat',Cat.color)
print(Cat.name)
```

Static Atrribute

```
class foo:
    all = 0 #靜態屬性
    def add(self):
        foo.all +=1

ins_1 =foo()
ins_2 = foo()
print(ins_1.all)
print(ins_2.all)
print(foo.all)
ins_1.add()
print(ins_1.all)
print(ins_2.all)
print(foo.all)
ins_1.all = 10
print(ins_1.all)
print(ins_2.all)
print(foo.all)
```

Public vs. Private

- Public: 任意處accessible
- Private: Class內accessible
- Code Stability
- Code Readability
- **Getter / Setter**

```python
class Cat:
    __color = 'gray'
    def __init__(self,name=''):
        self.name = name
    def get_color(self):
        print(self.__color)
    def set_color(self,color):
        self.__color = color
milk = cat('milk')
milk.get_color()
milk.set_color('brown')
milk.get_color()
print(milk.__color)
```

```python
class Cat:
    __color = 'gray'
    def __init__(self,name=''):
        self.name = name

    def letItSmell(self):
        self.__touch_cat()

    def __touch_cat(self):
        print('Meow~~')

milk = Cat('milk')
milk.letItSmell()
milk.__touch_cat()
```

Class Method vs. Instance Method vs. Static Method

- Instance Method: 透過self參數可以自由的存取Object的Attribute及其他Method，藉此來改變Object的狀態。

- Class Method: Python Class 中有@classmethod Decorator的Method，被呼叫時，相較於Instance Method的self參數指向物件(Object)，Class Method為cls參數，指向Class 但因它沒有self參數可以存取物件的屬性(Attribute)及方法(Method)，因此無法改變物件的狀態。

- Static Method: Python Class中有@staticmethod Decorator 的Method，可以接受任意的參數，也因為它沒有self及cls參數，所以Static Method無法改變Class及Object的狀態。好處:1. 在開發過程中可以避免新加入的開發者意外改變類別(Class)或物件(Object)的狀態，而影響到類別(Class)原始的設計。2. Static Method在類別中是獨立的，所以有助於單元的測試。

```python
class Cars:
    door = 4  # 類別屬性
    # 類別方法(Class Method)
    @classmethod
    def open_door(cls):
        print(f"{cls} has {cls.door} doors.")
mazda = Cars()
mazda.open_door()  #透過物件呼叫
Cars.open_door()  #透過類別呼叫
```

```python
#class method應用於產生物件
class Cars:
    # 建構式
    def __init__(self, seat, color):
        self.seat = seat
        self.color = color
    # 廂型車
    @classmethod
    def van(cls):
        return cls(6, "black")
    # 跑車
    @classmethod
    def sports_car(cls):
        return cls(4, "yellow")
van = Cars.van()
sports_car = Cars.sports_car()
```

```python
class Cars:

    @staticmethod # 靜態方法
    def speed_rate(distance, minute):
        return distance / minute
# 透過物件呼叫
van = Cars()
van_rate = van.speed_rate(10000, 20)
print("van rate: ", van_rate)
# 透過類別呼叫
sports_car_rate = Cars.speed_rate(20000, 20)
print("sports car rate: ", sports_car_rate)
```

Inheritance

- 子類別會擁有父類別公開的屬性(Attribute)及方法(Method)。
- 降低程式碼的重複性
- 方法覆寫(Method Overriding)
- 多層繼承(Multi-Level Inheritance)
- 多重繼承(Multiple Inheritance)

```python
class Car: # 繼承
    def __init__(self, wheels_number=4, car_doors=4, passengers=4):
        self.wheels_number = wheels_number
        self.car_doors = car_doors
        self.passengers = passengers

    def drive(self):
        print('driving the car.')

class SUV(Car):
    def __init__(self, wheels_number, car_doors, passengers, brand_name="", air_bag=2, sunroof=False):
        super().__init__(wheels_number, car_doors, passengers)
        self.brand_name = brand_name
        self.air_bag = air_bag
        self.sunroof = sunroof

    def drive(self):
        print('driving the SUV.') #Overriding

    def getDetails(self):
        print("Brand:", self.brand_name)
        print("Wheels number:", self.wheels_number)  # 可直接呼叫父類別的變數(屬性)
        print("Doors number:", self.car_doors)  # 可直接呼叫父類別的變數(屬性)
        print("Air-bags number:", self.air_bag)
        print("Sunroof:", self.sunroof)

toyota_rav = SUV(4, 5, 5, "Toyota RAV", 4, True)
toyota_rav.getDetails()
bmw_x5 = SUV(4, 5, 5, "BMW X5", 6, True)
bmw_x5.getDetails()
```

```python
class Vehicle: # 多層繼承
    def __init__(self, passengers=4):
        self.passengers = passengers

    def drive(self):
        print('driving the vehicle.')

class Car(Vehicle):
    def __init__(self, car_doors=4,wheels_number=4,driveMode='FF',passengers=4):
        super().__init__(passengers)
        self.car_doors = car_doors
        self.wheels_number = wheels_number
        self.driveMode = driveMode

    def drive(self):
        print('driving the car.') #Overriding

    def getDetails(self):
        print('passengers:', self.passengers)
        print("Wheels number:", self.wheels_number)
        print("Doors number:", self.car_doors)
        print("Drive mode:", self.driveMode)

class SUV(Car):
    def __init__(self, car_doors=4,wheels_number=4,driveMode='FF',passengers=4, airbags=2,sunroof=False):
        super().__init__(car_doors,wheels_number,driveMode,passengers)
        self.airbags = airbags
        self.sunroof = sunroof

    def getDetails(self):
        print('passengers:', self.passengers)
        print("Wheels number:", self.wheels_number)
        print("Doors number:", self.car_doors)
        print("Drive mode:", self.driveMode)
        print("Sunroof:", self.sunroof)
        print("Airbags:", self.airbags)

toyota_rav = SUV(4, 5, 5, "Toyota RAV", 4, True)
toyota_rav.getDetails()
```

```python
class Car: #多重繼承
    def __init__(self, wheels_number=4, car_doors=4, passengers=4):
        self.wheels_number = wheels_number
        self.car_doors = car_doors
        self.passengers = passengers

    def drive(self):
        print('driving the car.')
class Boat:
    def __init__(self, power=100, tonnage= 20):
        self.power = power
        self.tonnage = tonnage
    def drive(self):
        print('driving the boat.')

class Amphibian (Car,Boat):
    def __init__(self, wheels_number, car_doors, passengers, brand_name="", air_bag=2, sunroof=False, power=100, tonnage= 20):
        #super().__init__(wheels_number, car_doors, passengers)
        Car.__init__(self,wheels_number, car_doors, passengers) # 個別呼叫父類別的__init__
        Boat.__init__(self,power,tonnage) # 個別呼叫父類別的__init__
        self.brand_name = brand_name
        self.air_bag = air_bag
        self.sunroof = sunroof

    def getDetails(self):
        print("Brand:", self.brand_name)
        print("Wheels number:", self.wheels_number)  # 可直接呼叫父類別的變數(屬性)
        print("Doors number:", self.car_doors)  # 可直接呼叫父類別的變數(屬性)
        print("Air-bags number:", self.air_bag)
        print("Sunroof:", self.sunroof)
        print("car_doors:", self.car_doors)
        print("tonnage:", self.tonnage)

toyota_rav = Amphibian(4, 5, 5, "Toyota RAV", 4, True)
toyota_rav.getDetails()
toyota_rav.drive() # 先檢查子類別有沒有此method若有則執行，若無則檢查第一個父類別有沒有此method…依此類推
```

Class的特殊函數(方法)

```
class Rational:
def __init__(self, n, d): # 物件建立之後所要建立的初始化動作
self.numer = n
self.denom = d

def __str__(self): # 定義物件的字串描述
return str(self.numer) + '/' + str(self.denom)

def __add__(self, that): # 定義 + 運算
return Rational(self.numer * that.denom + that.numer * self.denom,
self.denom * that.denom)

def __sub__(self, that): # 定義 - 運算
return Rational(self.numer * that.denom - that.numer * self.denom,
self.denom * that.denom)

def __mul__(self, that): # 定義 * 運算
return Rational(self.numer * that.numer,
self.denom * that.denom)

def __truediv__(self, that): # 定義 / 運算
return Rational(self.numer * that.denom,
self.denom * that.denom)

def __eq__(self, that): # 定義 == 運算
return self.numer * that.denom == that.numer * self.denom
```

```python
x = Rational(1, 2)
y = Rational(2, 3)
z = Rational(2, 3)
print(x) # 1/2
print(y) # 2/3
print(x + y) # 7/6
print(x - y) # -1/6
print(x * y) # 2/6
print(x / y) # 3/6
print(x == y) # False
print(y == z) # True
```

```python
class Some:
    def __init__(self):
        self.inner = {}

    def __setitem__(self, name, value):
        self.inner[name] = value

    def __getitem__(self, name):
        return self.inner[name]

    def __func1(self,i = 0):
        ks = list(self.inner.keys())
        print(ks[i])

    def __call__(self,i = 0):
        self.__func1(i)
s = Some()
s[0] = 100
s['Thanos'] = 'GOOD MAN'
s['Justin'] = 'Message'
print(s[0])
print(s['Justin'])
s(1)
```

类别	方法名
字符串/字节序列表示形式	__repr__, __str__, __format__, __bytes__
数值转换	__abs__, __bool__, __complex__, __int__, __float__, __hash__, __index__
集合模拟	__len__, __getitem__, __setitem__, __delitem__, __contains__
迭代枚举	__iter__, __reversed__, __next__
可调用模拟	__call__
上下文管理	__enter__, __exit__
实例创建和销毁	__new__, __init__, __del__
属性管理	__getattr__, __setattr__, getattribute__, __setattribute__, __delattr__, __dir__
属性描述符	__get__, __set__, __delete__
跟类相关的服务	__prepare__, __instancecheck__, __subclasscheck__

类别	方法名和对应的运算符
一元运算符	__neg__ - , __pos__ +, __abs__ abs()
众多比较运算符	__lt__ <, __le__ <=, __eq__ =, __ne__ !=, __gt__ >, __ge__ >=
算术运算符	__add__ +, __sub__ - , __mul__ *, __truediv__ /, __floordiv //, __mod__ %, __divmod__ divmod(), __pow__ **或pow(), __round__ round()
反向算术运算符	__radd__, __rsub__, __rmul__, __rtruediv__, __rfloordiv__, __rmod__, __rdivmod__, __rpow__
增量赋值算术运算符	__iadd__, __isub__, __imul__ , __itruediv__, __ifloordiv__, __imod__, __ipow__
位运算符	__invert__ ~, __lshift__ <<, __rshift__ >>, __and__ &, __or__ |, __xor__ ^
反向位运算符	__rlshift__, __rrshift__, __rand__, __rxor__, __ror__
增量赋值位运算符	__ilshift__, __irshift__, __iand__, __ixor__, __ior__

Decorator

```python
def print_func_name(func):
    def wrap():
        print("Now use function '{}'".format(func.__name__))
        func()
    return wrap

def dog_bark():
    print("Bark !!!")

def cat_miaow():
    print("Miaow ~~~")

if __name__ == "__main__":
    print_func_name(dog_bark)()
    # > Now use function 'dog_bark'
    # > Bark !!!

    print_func_name(cat_miaow)()
    # > Now use function 'cat_miaow'
    # > Miaow ~~~
```

Decorator-syntax candy

```python
def print_func_name(func):
    def warp():
        print("Now use function '{}'".format(func.__name__))
        func()
    return warp

@print_func_name # syntax candy
def dog_bark():
    print("Bark !!!")

@print_func_name
def cat_miaow():
    print("Miaow ~~~")

if __name__ == "__main__":
    dog_bark()
    # > Now use function 'dog_bark'
    # > Bark !!!

    cat_miaow()
    # > Now use function 'cat_miaow'
    # > Miaow ~~~
```

Decorator – 順序

```
def print_func_name(func):
    def warp_1():
        print("Now use function '{}'".format(func.__name__))
        func()
    return warp_1

def print_time(func):
    import time
    def warp_2():
        print("Now the Unix time is
{}".format(int(time.time())))
        func()
    return warp_2

@print_func_name
@print_time
def dog_bark():
    print("Bark !!!")

if __name__ == "__main__":
    dog_bark()
    # > Now use function 'warp_2'
    # > Now the Unix time is 1541239747
    # > Bark !!!
```

Decorator with parameter

```
import time

def print_func_name(time):
    def decorator(func):
        def wrap():
            print("Now use function
'{}'".format(func.__name__))
            print("Now Unix time is
{}.".format(int(time)))
            func()
        return wrap
    return decorator

@print_func_name(time=(time.time()))
def dog_bark():
    print("Bark !!!")

if __name__ == "__main__":
    dog_bark()
    # > Now use function 'dog_bark'
    # > Now Unix time is 1639491313.
    # > Bark !!!
```

Class Decorator

```python
class Dog:
    def __init__(self, func):
        self.age = 10
        self.talent = func

    def bark(self):
        print("Bark !!!")

@Dog
def dog_can_pee():
    print("I can pee very hard......")

if __name__ == "__main__":
    dog = dog_can_pee

    print(dog.age)
    # > 10

    dog.bark()
    # > Bark !!!

    dog.talent()
    # > I can pee very hard......
```

```python
class Dog:
    def __init__(self, func):
        self.talent = func

    def bark(self):
        print("Bark !!!")

@Dog
def dog_can_pee():
    print("I can pee very hard.....")

@Dog
def dog_can_jump():
    print("I can jump uselessly QQQ")

@Dog
def dog_can_poo():
    print("I can poo like a super pooping machine!")

if __name__ == "__main__":
    dog_1 = dog_can_pee
    dog_1.talent()
    # > I can pee very hard.....

    dog_2 = dog_can_jump
    dog_2.talent()
    # > I can jump uselessly QQQ

    dog_3 = dog_can_poo
    dog_3.talent()
    # > I can poo like a super pooping machine!
```

練習

- 定義一個**角色**的 Class，情境如下：
  - 角色有名字、血量和攻擊力基本屬性。
  - 角色可以確認攻擊目標。
  - 角色攻擊完敵人後，顯示敵人剩餘血量。
- 另外包裝一個 battle() 函式，將英雄攻擊怪物的流程包裝在內，情境如下：
  - 英雄先將怪物視為攻擊目標，然後攻擊怪物。
  - 怪物被攻擊後，將英雄視為攻擊目標。
  - 英雄會攻擊兩次，怪物只攻擊一次。