

EXT



pythonTM

Class II

張傑帆 Chang, Jie-Fan



NTU CSIE

OUTLINE

■ 類別II



屬性封裝

- Python 類別 (class) 的屬性 (attribute) 權限預設是公開的，因此類別以外的地方也可以存取，例如

```
class Demo:
```

```
    x = 0
```

```
    def __init__(self, i):
```

```
        self.i = i
```

```
        Demo.x += 1
```

```
    def hello(self):
```

```
        print("hello", self.i)
```

```
a = Demo("Tom")
```

```
a.hello()
```

```
print("hello", a.i)
```

```
print()
```

```
print("a.i =", a.i)
```

```
print("Demo.x =", Demo.x)
```



受保護屬性

在屬性的前面加上雙底線「__」即變成受保護的屬性

```
class Demo:
```

```
    __x = 0
```

```
    def __init__(self, i):
```

```
        self.__i = i
```

```
        Demo.__x += 1
```

```
    def hello(self):
```


```
        print("hello", self.__i)
```

```
a = Demo("Tom")
```


```
a.hello()
```

```
print("Demo.x =", Demo.__x) #受保護而不能存取
```





```
>>>
hello Tom
Traceback (most recent call last):
  File "D:\Dropbox\台大課程\系統訓練班\Python\PPT\ex\class2\
p02.py", line 11, in <module>
    print("a.x =", a.__x)  #受保護而不能存取
AttributeError: 'Demo' object has no attribute '__x'
```



類別方法

- 回傳受保護屬性，類別方法需要一個特別的參數 (parameter)，習慣上使用 **cls**，這與實體方法的 **self** 類似，不同的是 **cls** 用來存取類別的屬性 (attribute)

@classmethod

def function_name(cls):

return cls.__變數



class Demo:

```
    __x = 0  
    def __init__(self, i):  
        self.__i = i  
        Demo.__x += 1  
    def hello(self):  
        print("hello", self.__i)
```

```
    @classmethod  
    def getX(cls):  
        return cls.__x
```

```
a = Demo("Tom")  
a.hello()  
print("Demo.x =", Demo.getX())
```

- 方法預設也都是公開的，若要定義私有的方法，也就是只能在類別內呼叫的方法，
- 同樣在方法識別字名稱前加上連續兩個底線符號，這樣的方法就變成私有的



類別繼承

- 若定義了很多類別 (class)，這些類別中又具有相當多相同的屬性 (attribute) 或方法 (method) 定義，這時候，可利用 Python 的繼承 (inheritance) 機制
- 將共通的屬性及方法提取出來，另行定義父類別 (superclass)，然後將原本提出共通屬性及方法改為繼承 (inherit) 父類別的子類別 (subclass)。



類別繼承

- 繼承的格式如下

```
class SubDemo(Demo):  
    #dosomething
```

- 這是從 **SubDemo** 類別去繼承 **Demo**，注意類別名稱後的小括弧中註明父類別。



class Demo:

```
    __x = 0
```

```
    def __init__(self, i):
```

```
        self.__i = i
```

```
        Demo.__x += 1
```

```
    def hello(self):
```

```
        print("hello", self.__i)
```

```
    @classmethod
```

```
    def getX(cls):
```

```
        return cls.__x
```

```
    @classmethod
```

```
    def add(cls):
```

```
        Demo.__x += 1
```

class subDemo(Demo):

```
    pass #略過
```

```
a = Demo("Tom")
```

```
a.hello()
```

```
b = subDemo("John")
```

```
b.hello()
```

```
print("Demo.x =", Demo.getX())
```



內建函數

- 內建函數 (function) `isinstance()` 可以判斷某一個物件是否為某一個類別所建構的實體 (instance)，若真則回傳 **True**，否則回傳 **False**。
- 另一個內建函數 `issubclass()` 則可以判斷某一個類別是否為另一個類別的子類別，同樣的，若真則回傳 **True**，否則回傳 **False**。




```
a = Demo("Tom")  
b = subDemo("John")
```

```
#instance
```

```
print(isinstance(a, Demo))  
print(isinstance(a, subDemo))  
print(isinstance(b, Demo))  
print(isinstance(b, subDemo))
```

```
#issubclass
```

```
print(issubclass(subDemo, Demo))  
print(issubclass(Demo, subDemo))
```

```
True  
False  
True  
True  
True  
False  
>>>
```

變數 (variable) **b** 雖然是由 **SubDemo** 建立的，但是 **b** 也會是 **Demo** 的實體，這是由於物件實體的建構過程中，會先建立父類別的部份，因此也會建立屬於 **b** 的父類別物件實體，使 **b** 得以運用父類別的屬性及方法。



子類別 方法改寫

- 子類別 (subclass) 可依本身特性設定自己的屬性 (attribute) 與方法 (method)，也會從父類別 (superclass) 繼承 (inherit) 屬性與方法。一般來說，沒有設定成私有的屬性及方法都會被繼承，子類別可由父類別公開的方法存取父類別私有的屬性。
- 子類別也可依需要改寫 (override) 父類別的方法，這是說子類別需要用到與父類別具有相同名稱的方法，但是子類別需要的功能有所修改、擴充或增加，因此當子類別裡頭定義與父類別相同名稱的方法時，就會改寫父類別的方法。經過改寫，子類別的方法完全屬於子類別所有。



class Demo:

__x = 0

def __init__(self, i):

self.__i = i

Demo.__x += 1

def hello(self):

print("hello", self.__i)

@classmethod

def getX(cls):

return cls.__x

@classmethod

def add(cls):

Demo.__x += 1

class subDemo(Demo):

def __init__(self, i, j):

self.__i = i

self.__j = j

def hello(self):

print("hello", self.__i, self.__j)




```
a = Demo("Tom")
a.hello()
b = subDemo("John","Mary")
b.hello()
print("Demo.x =", Demo.getX())
```

```
>>>
hello Tom
hello John Mary
Demo.x = 1
>>> |
```

- Demo 為父類別，定義四個方法，SubDemo 為子類別，改寫 Demo 的兩個方法，包括 `__init__()` 與 `hello()`。



SUPER() — 呼叫父類別的方法

- 利用內建函數 (function) `super()`，呼叫 (call) 父類別的方法

```
class subDemo(Demo):
```

```
    def __init__(self, i, j="guest"):
```

```
        self.__i = i
```

```
        self.__j = j
```

```
    def hello(self):
```

```
        print("hello", self.__i, self.__j)
```

```
    def superHello(self):
```

```
        super().__init__(self.__i)
```

```
        super().hello()
```

```
a = Demo("Tom")
```

```
a.hello()
```

```
b = subDemo("John", "Mary")
```

```
b.hello()
```

```
print("Demo.x =", Demo.getX())
```

```
b.superHello()
```

```
>>>
hello Tom
hello John Mary
Demo.x = 1
hello John
>>> |
```



類別 多重繼承

- 設計類別 (class) 時，父類別 (superclass) 可以有**多個**，這是說子類別 (subclass) 能夠繼承 (inherit) 多個父類別，使子類別可以有**多種特性**。
- 這裡須注意一點，當子類別繼承 (inheritance) 超過一個來源的時候，會以寫在**最左邊的父類別優先繼承**，這是說，多個父類別**如果有相同名稱的屬性 (attribute) 與方法 (method)**，例如 `__init__()`、`__str__()` 等，就會以**最左邊的父類別優先**。



#多重繼承

class Demo:

__x = 0

def __init__(self, i):

self.__i = i

Demo.__x += 1

def hello(self):

print("hello", self.__i)

@classmethod

def getX(cls):

return cls.__x

@classmethod

def add(cls):

Demo.__x += 1

class Demo2:

def __init__(self, i):

self.__i = i

def reverseString(self, string):

reverse=""

for i in range(len(string)-1, -1, -1):

reverse += string[i]

return reverse



```

class subDemo(Demo2,Demo):
    def __init__(self, i, j="guest"):
        super().__init__(i)
        self.__i = i
        self.__j = j
    def hello(self):
        print("hello", self.__i,self.__j)
    def superHello(self):
        super().__init__(self.__i)
        super().hello()

```

```

a = subDemo("Tom")
print(a.reverseString("Tom"))
print("Demo.x =", Demo.getX())

```

```

>>>
moT
Demo.x = 0
>>>

```

```

>>> a.add()
>>> Demo.getX()
1
>>> a.getX()
1
>>>

```



類別 `__DEL__()` 解構子

- 建構子 (constructor) 用來建立物件 (object) ，當物件不需要被使用時，直譯器 (interpreter) 會主動替物件呼叫 `__del__()` 方法 (method) ，這是物件自動銷毀的方法，也就是從記憶體中釋放空間的步驟，被稱為解構子 (destructor) ，當然，我們也可以改寫 (override) 這個方法。




```
class Demo:
```

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

```
    def __str__(self):  
        return str(self.i)
```

```
    def __del__(self):  
        print("del called: " + self.__str__())
```

```
    def hello(self):  
        print("hello " + self.__str__())
```

```
a = Demo("Tommy")
```

```
a.hello()
```

```
>>>
```

```
hello Tommy
```

```
>>>
```



解構子

- 我們只有使用變數 (variable) **a** 一個名稱，利用建構子 **Demo()** 建立物件後呼叫 **hello()**，然後重新呼叫 **Demo()** 建立另一個 **Demo** 型態的物件，我們可以看到直譯器主動呼叫 **__del__()**，印出 "del called" 的訊息。

```
>>>  
hello Tommy  
>>> a=Demo("gg")  
del called: Tommy  
>>> a=1  
del called: gg  
>>> a=Demo("cc")  
>>> del a  
del called: cc  
>>>
```

- 最後程式結束執行前，直譯器同樣主動呼叫最後建立物件解構子，完全釋放所使用的記憶體空間。



類別 多型

- 多型 (polymorphism) 是物件導向程式語言 (object-oriented programming language) 的一項主要特性，使物件 (object) 的使用更具彈性。
- 比如有動物 (Animal) 之類別 (Class)，而且由動物繼承出類別雞 (Chicken) 和類別狗 (Dog)，並對同一源自類別動物 (父類別) 之一訊息有不同的響應，如類別動物有「叫()」之動作，而類別雞會「啼叫()」，類別狗則會「吠叫()」，則稱之為多型。
- 簡單來說，多型可使物件的型態具有通用的效力，例如以下程式




```
class Demo:
```

```
    def __init__(self, i):
```

```
        self.i = i
```

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

```
        return str(self.i)
```

```
    def hello(self):
```

```
        print("hello " + self.__str__())
```

```
class SubDemo1(Demo):
```

```
    def __init__(self, i, j):
```

```
        super().__init__(i)
```

```
        self.j = j
```

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

```
        return super().__str__() + str(self.j)
```

```
>>>
```

```
hello 2233
```

```
hello 4455
```

```
>>>
```



```
class SubDemo2(Demo):
```

```
    def __init__(self, i, j):
```

```
        super().__init__(i)
```

```
        self.j = j
```

```
        self.k = str(self.i) + str(self.j)
```

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

```
        return self.k
```

```
a = SubDemo1(22, 33)
```

```
b = SubDemo2(44, "55")
```

```
a.hello()
```

```
b.hello()
```

```
>>>
```

```
hello 2233
```

```
hello 4455
```

```
>>>
```



其他的例子

d1 = "12345"

d2 = [1, 2, 3, 4, "5"]

print(d1.count("4"))

print(d2.count("4"))

```
>>> d1 = '12345'  
>>> d2 = [1, 2, 3, 4, '5']  
>>> d1.count("4")  
1  
>>> d2.count("4")  
0
```



其他的例子

- d1 為字串 (string) ， d2 為串列 (list) ，兩者皆屬於序列 (sequence) 的複合資料型態 (compound data type) ，有通用的 **count()** 方法，可計算某元素 (element) 累計出現的次數。
- 多型的應用很多，例如串列中可接受不同型態的物件當元素，或是方法可用不同型態的參數等。



課堂練習

試建立下列類別:

- **person**

name, gender, age

getGender(), getAge()

sayHello() 會印出 hello, 我是xxx, 性別, 現年幾歲

- **student**

addCourse(課程物件): 表示student有修某堂課程

addGrades(課程, 分數): 加入成績於某堂課程

removeCourse(課程物件): 表示student停修某堂課程

avg(課程物件): 算出某堂課程的平均分數

fcount(課程物件): 算出某堂課程被當掉的總數



■ teacher

addCourse(課程物件): 表示teacher有教某堂課程

removeCourse(課程物件): 表示teacher停教某堂課程

listStudents(): 印出所有被教的學生名單，並以成績排序

listNoPass(): 印出所有平均不及格的同學

■ course

name, teacher, students[]

listMembers(): 印出所有學生及老師名單

avg(): 印出修此科目的所有學生平均

sayHello(): 印出 hello, 我是xxx, 教yyy課程

student, teacher 繼承 person



#student

s1 = student("Tom","M","20")

s2 = student("Jane","F","21")

s3 = student("John","M","21")

s4 = student("Ann","F","19")

s5 = student("Peter","M","20")

#teacher

t1 = teacher("JieFan","M","29")

t2 = teacher("Mary","F","26")

#course

c1 = course('python')

c2 = course('c++')

c3 = course('Java')



t1.addCourse(c1)

t1.addCourse(c2)

t2.addCourse(c3)

s1.addCourse(c1)

s2.addCourse(c1)

s3.addCourse(c1)

s2.addCourse(c2)

s3.addCourse(c2)

s4.addCourse(c2)

s1.addGrades(c1,100)

s1.addGrades(c1,80)

s2.addGrades(c1,30)

s2.addGrades(c1,49)

s3.addGrades(c1,66)

s3.addGrades(c1,90)

s2.addGrades(c2,88)

s2.addGrades(c2,65)

s3.addGrades(c2,30)

s3.addGrades(c2,88)

s4.addGrades(c2,47)

s4.addGrades(c2,98)

t1.listStudents()

t1.listNoPass()

c1.listMembers()

c1.avg()

c2.avg()

c1.sayHello()



>>>

python課程:

[('Tom', 90.0), ('John', 78.0), ('Jane', 39.5)]

c++課程:

[('Jane', 76.5), ('Ann', 72.5), ('John', 59.0)]

python課程:

[('Jane', 39.5)]

c++課程:

[('John', 59.0)]

授課老師: JieFan

學生:

Tom

Jane

John

69.16666666666667

69.33333333333333

helo, 我是 JieFan, 教python課程

>>>




```
# -*- coding: utf-8 -*-
class person:
    def __init__(self, name, gender, age):
        self.name = name
        self.__gender = gender
        self.__age = age
    def getGender(self):
        return self.__gender
    def getAge(self):
        return self.__age
    def sayHello(self):
        print('hello, i am %s, %s, %s years old.'
              %(self.name, self.__gender, self.__age))
```



```
class student(person):
    def __init__(self, name, gender, age):
        super().__init__(name, gender, age)
        self.grades=[]

    def addCourse(self, course):
        course.students.append(self)

    def addGrades(self, course, grade):
        self.grades.append((course, grade))

    def removeCourse(self, course):
        course.students.remove(self)

    def avg(self, course):
        sum_grades = 0
        total = 0
        for i in self.grades:
            if course == i[0]:
                sum_grades += i[1]
                total += 1
        return sum_grades/total

    def fcount(self, course):
        fcount=0
        for i in self.grades:
            if course == i[0]:
                if i[1] < 60:
                    fcount +=1
        return fcount
```



[illegible]


```
class course:
    def __init__(self,c_name):
        self.c_name=c_name
        self.teacher=''
        self.students=[]
    def listMembers(self):
        print('授課老師:%s'%self.teacher.name)
        print('學生:')
        for s in self.students:
            print(s.name)
    def avg(self):
        sum_avg=0
        for s in self.students:
            sum_avg+=s.avg(self)
        print(sum_avg/len(self.students))
    def sayHello(self):
        print('hello,我是 %s,教%s課程'%(self.teacher.name,self.c_name))
```

