
Python程式設計

類別(Class)

高師大數學系

葉倚任

模組(Modules)

還記得之前寫了一個 bank 的模組嗎？

所有的操作都跟帳戶狀態相關，那可以將這些屬性與動作組織再起一起形成一個類別嗎？

```
bank.py
1 def account(name, number, balance):
2     return {'name': name, 'number': number, 'balance': balance}
3
4 def deposit(acct, amount):
5     if amount <= 0:
6         print('存款金額不得為負')
7     else:
8         acct['balance'] += amount
9
10 def withdraw(acct, amount):
11     if amount > acct['balance']:
12         print('餘額不足')
13     else:
14         acct['balance'] -= amount
15
16 def desc(acct):
17     return 'Account:' + str(acct)
18
```

← 建立帳戶

← 存款

← 提款

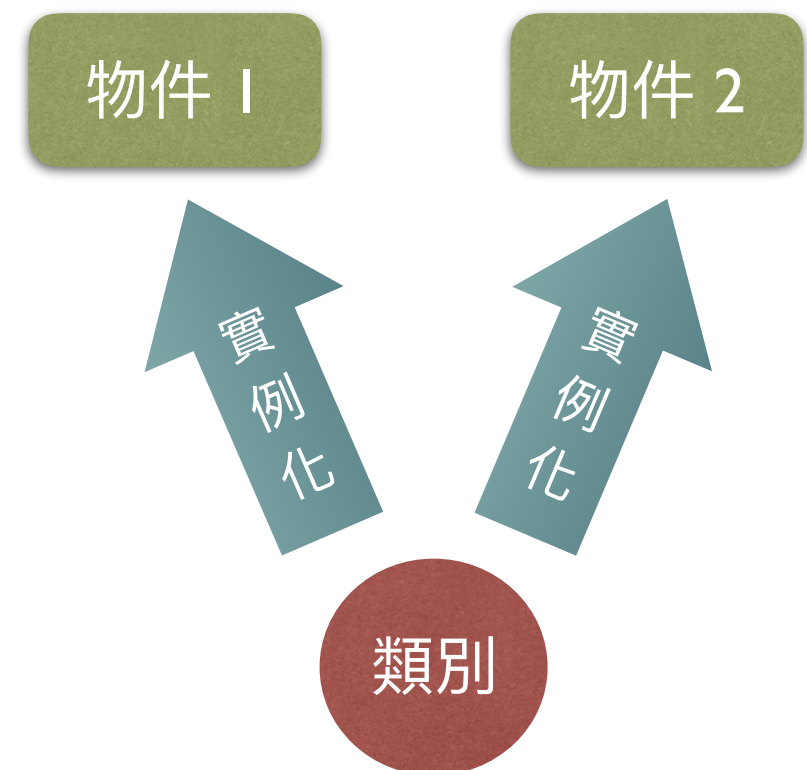
← 描述帳戶狀態

類別(class)

- 每個類別都有自己的屬性和方法
- 類別的屬性其實就是類別內部的變數
- 類別的方法則是類別內部定義的函數

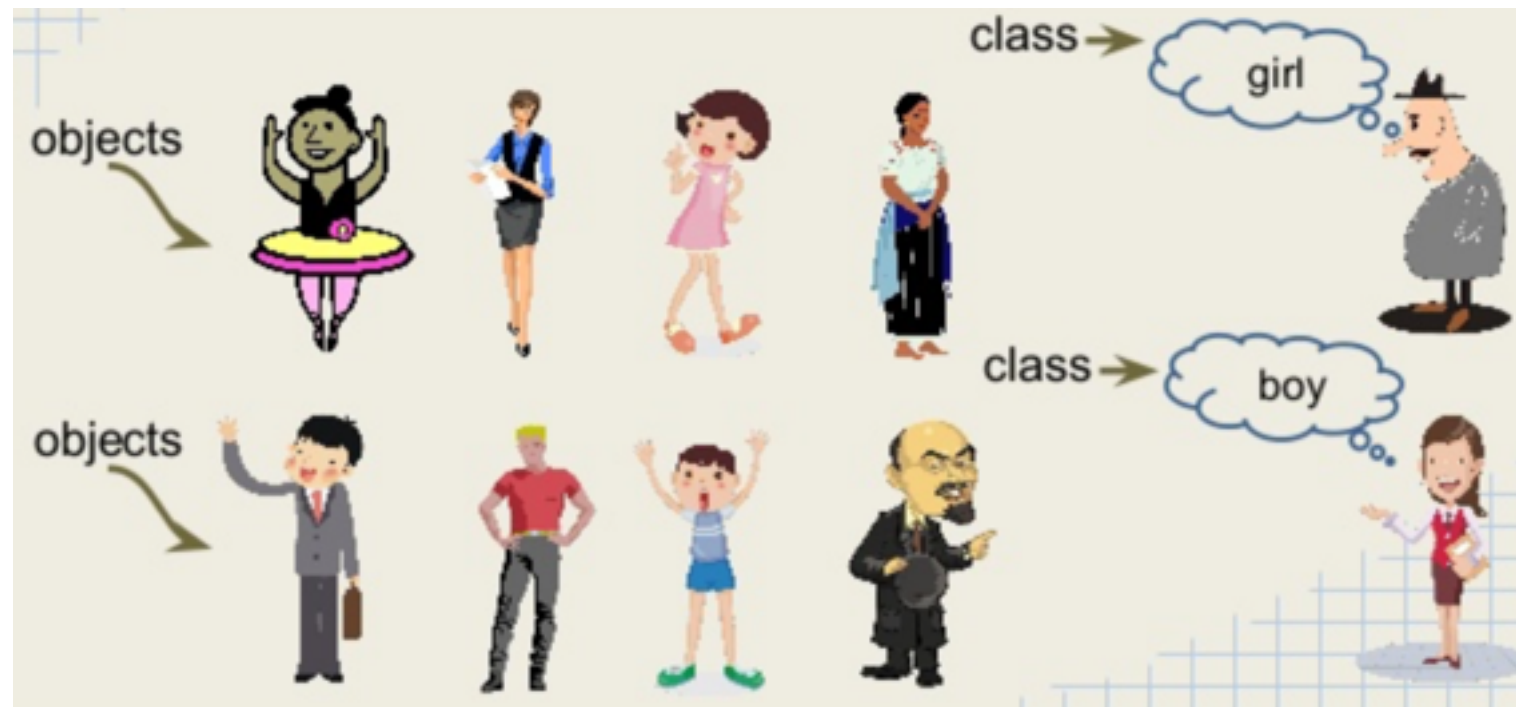
類別(class)與物件(object)

- 物件是類別產生的實體結果
- 每個物件的屬性值可能不同，但由同一種類別產生實體得來的物件，都擁有共同的屬性和方法



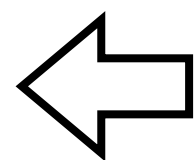
物件(object)

物件導向程式設計中的每一個物件都應該能夠接受資料、處理資料並將資料傳達給其它物件，因此它們都可以被看作一個小型的「機器」，或者說是負有責任的角色。



類別的定義與使用

```
class <類別名稱> :  
    <敘述 1>  
    <敘述 2>  
    ...
```



在敘述中定義類別的屬性與方法

定義類別屬性

OOexample.py

```
1 class book:
2     author = ''
3     bname = ''
4     page = 0
5     price = 0
6
```

```
import OOexample

a = OOexample.book()
print(a.author)
a.author = 'Tom'
print(a.author)
```

Tom

物件與物件之間不會互相影響

```
import OOexample
```

```
mybook = OOexample.book()
```

```
mybook.author = 'Tom'
```

```
urbook = OOexample.book()
```

```
urbook.author = 'John'
```

```
print(mybook.author)
```

```
print(urbook.author)
```

```
print(mybook)
```

```
print(urbook)
```

Tom

John

```
<OOexample.book object at 0x1043471d0>
```

```
<OOexample.book object at 0x1043470b8>
```

← 利用 book 這個 class 產生 mybook 這個實例

← 利用 book 這個 class 產生 urbook 這個實例

← 利用 mybook 跟 urbook 互不相影響

← mybook 跟 urbook 分別是在不同記憶體體的物件

直接對類別屬性做操作的注意事項

```
import OOexample

hisbook = OOexample.book()
print(hisbook.price)
OOexample.book.price = 100
herbook = OOexample.book()
print(herbook.price)
print(hisbook.price)
```

0
100
100

類別屬性可以直接用類別名稱來進行屬性操作，但是這樣會影響到其他物件的初始狀態。

```
import OOexample

hisbook = OOexample.book()
hisbook.price = 50
print(hisbook.price)
OOexample.book.price = 100
herbook = OOexample.book()
print(herbook.price)
print(hisbook.price)
```

50
100
50 ← 已經不是初始狀態所以不影響

定義類別方法

OObank.py

```
1 class bank:
2     name=''
3     number=''
4     balance=0
5
6     def deposit(self,amount):
7         if amount <= 0:
8             print('存款金額不得為負')
9         else:
10            self.balance += amount
11
12    def withdraw(self,amount):
13        if amount > self.balance:
14            print('餘額不足')
15        else:
16            self.balance -=amount
17
18    def desc(self):
19        return "Account('{0}','{1}','{2}').format(
20            self.name,self.number,self.balance
21        )
```

← 定義類別方法跟之前定義函數一樣

在 python 中，類別方法的第一個參數一定是物件本身 (self)，如果其他參數，可以從第二個參數開始依序定義。

使用範例

```
import OObank
```

```
acct = OObank.bank()
```

```
print(acct.balance)
```

```
acct.deposit(100)
```

```
print(acct.balance)
```

```
acct.name = 'Tom'
```

```
acct.number = '9527'
```

```
print(acct.desc())
```

```
0
```

```
100
```

```
Account('Tom', '9527', '100')
```

← 一樣利用 bank 這個 class 產生 acct 這個實例

← 實例使用 deposit 這個類別方法

注意：使用方法時，會自動把物件本身當做第一個參數，所以不用再給物件之名稱

定義物件初始化方法 `__init__()`

- 可以將初始化流程，使用 `__init__()` 方法定義在類別之中
- 方法前後各有兩個連線底線，為類別的專有方法，可利用特定函數呼叫此方法

```
2     def __init__(self, name, number, balance):  
3         self.name = name  
4         self.number = number  
5         self.balance = balance
```

```
import bankClass
```

```
acct = bankClass.Account('John', '9527', 100)
```

定義物件描述字串方法 `__str__()`

- `__str__()` 為傳回物件描述字串的方法
- 可利用 `print()` 或者 `str()` 來呼叫此專有方法

```
18     def __str__(self):  
19         return "Account('{0}','{1}','{2}').format(  
20             self.name,self.number,self.balance  
21         )
```

```
import bankClass  
acct = bankClass.Account('John','9527',100)  
acct.deposit(168)  
print(acct)
```

```
Account('John','9527','268')
```

完整之 Account class

bankClass.py

```
1 class Account:
2     def __init__(self, name, number, balance):
3         self.name = name
4         self.number = number
5         self.balance = balance
6
7     def deposit(self, amount):
8         if amount <= 0:
9             print('存款金額不得為負')
10        else:
11            self.balance += amount
12
13    def withdraw(self, amount):
14        if amount > self.balance:
15            print('餘額不足')
16        else:
17            self.balance -= amount
18    def __str__(self):
19        return "Account('{0}', '{1}', '{2}').format(
20            self.name, self.number, self.balance
21        )
```

類別的私有屬性

- 類別屬性可分為公有屬性與私有屬性
 - 公有屬性：此屬性可以被外部存取
 - 私有屬性：可以用 `__xxx` 來定義私有屬性，只允許類別內部使用

```
OOexampleP.py
1 class book:
2     __author = ''
3     __bname = ''
4     __page = 0
5     price = 0
6
```

```
import OOexampleP

theirbook = OOexampleP.book()
print(theirbook.price)
print(theirbook.page)

0

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-1-82ac11572e55> in <module>()
      3 theirbook = OOexampleP.book()
      4 print(theirbook.price)
----> 5 print(theirbook.page)

AttributeError: 'book' object has no attribute 'page'
```


回到 bank 之例子

bankClassP.py

```
1 class Account:
2     def __init__(self, name, number, balance):
3         self.__name = name
4         self.__number = number
5         self.__balance = balance
6
7     def deposit(self, amount):
8         if amount <= 0:
9             print('存款金額不得為負')
10        else:
11            self.__balance += amount
12
13    def withdraw(self, amount):
14        if amount > self.__balance:
15            print('餘額不足')
16        else:
17            self.__balance -= amount
18    def __str__(self):
19        return "Account('{0}', '{1}', '{2}')" .format(
20            self.__name, self.__number, self.__balance
21        )
```


定義外部屬性

- 基本上，可以直接定義一些方法來傳回內部屬性的值

```
def number(self):  
    return self.__number
```

@property

```
def balance(self):  
    return self.__balance
```

```
import bankClassP  
  
acct = bankClassP.Account('Ian', '9527', 100)  
print(acct)  
print(acct.number)  
  
Account('Ian', '9527', '100')  
<bound method Account.number of <bankClassP.Account object at 0x10422f748>>
```


```
import bankClassP  
  
acct2 = bankClassP.Account('Ian', '9527', 100)  
print(acct2)  
print(acct2.balance)  
  
Account('Ian', '9527', '100')  
100
```

綁定方法(bound method)

```
import bankClassP

acct3 = bankClassP.Account('Ian', '9527', 100)
deposit = acct3.deposit
print(deposit)
deposit(500)
print(acct3)
```

```
<bound method Account.deposit of <bankClassP.Account object at 0x1042739b0>>
Account('Ian', '9527', '600')
```



表示此函式是一個綁定方法，已經綁定一個Account 之實例 (程式碼裡面的acct3)

靜態方法

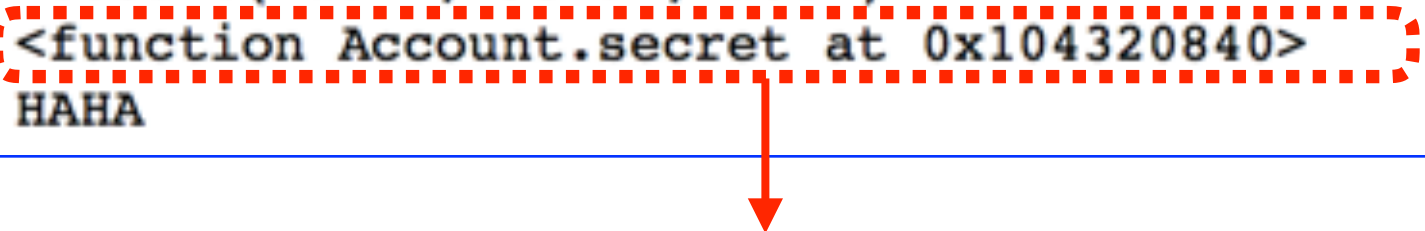
- 在定義類別時，希望某個方法不被拿來作為綁定方法，**可以使 @staticmethod**

```
@staticmethod
def secret(mystr):
    print(mystr)
```

```
import bankClassP

acct4 = bankClassP.Account('Ian', '9527', 100)
print(acct4)
secret = bankClassP.Account.secret
print(secret)
secret('HAHA')
```

```
Account('Ian', '9527', '100')
<function Account.secret at 0x104320840>
HAHA
```



- 呼叫此方法時，實例不會傳入靜態方法裡面
- 建議使用類別名稱來呼叫靜態方法

屬性名稱空間

每個物件本身，都會有個 `__dict__` 屬性，當中記錄著類別或實例所擁有的特性：

```
import bankClassP

acct5 = bankClassP.Account('Ian', '9527', 100)
print(acct5.__dict__)

{'_Account__balance': 100, 'name': 'Ian', '_Account__number': '9527'}
```

若想取得 `__dict__` 的資料，其實可以使用 `vars()` 函式：

```
import bankClassP

acct5 = bankClassP.Account('Ian', '9527', 100)
print(vars(acct5))

{'_Account__balance': 100, 'name': 'Ian', '_Account__number': '9527'}
```

定義運算子

- 目前我們有接觸了 `__init__()` 與 `__str__()` 這兩個特別方法，在 python 中是用來定義特定行為。
- 型態彼此間的運算行為也是用特別方法來定義的

```
x=10
y=3
print(x+y)
print(x.__add__(y))
```

13

13

範例：有理數類別與定義其運算子

xmath.py

```
1 class Rational:
2     def __init__(self, numer, denom):
3         self.numer = numer
4         self.denom = denom
5
6     def __add__(self, that):
7         return Rational(
8             self.numer * that.denom + that.numer * self.denom,
9             self.denom * that.denom
10        )
11
12    def __sub__(self, that):
13        return Rational(
14            self.numer * that.denom - that.numer * self.denom,
15            self.denom * that.denom
16        )
17
18    def __mul__(self, that):
19        return Rational(
20            self.numer * that.numer,
21            self.denom * that.denom
22        )
23
```

```
24    def __truediv__(self, that):
25        return Rational(
26            self.numer * that.denom,
27            self.denom * that.denom
28        )
29
30    def __str__(self):
31        return '{numer}/{denom}'.format(
32            numer = self.numer, denom = self.denom
33        )
```

其他運算子

- 想定義 $>$ 、 $>=$ 、 $<$ 、 $<=$ 、 $==$ 、 $!=$ 等比較，可以分別實作
 - $>$: `__gt__()`
 - $>=$: `__ge__()`
 - $<$: `__lt__()`
 - $<=$: `__le__()`
 - $==$: `__eq__()`
 - $!=$: `__ne__()`

練習

- 額外定義有理數類別之 $>$ 、 $<$ 、 $=$ 之比較

類別實例建構方法：__new__()

- __init__()方法是定義物件建立後初始化的流程，也就是執行到__init__()方法時，物件實際上已建構完成。
- 傳入__init__()的引數，並不是作為建構物件之用，而是作為初始物件之用。
- 實際上要決定如何建構物件，必須定義__new__()方法。

__new__()測試

- __new__()這個方法的第一個參數總是傳入類別本身，之後可接任意參數作為建構物件之用。
- __new__() 方法可以傳回物件，如果傳回的物件是第一個參數的類別實例，則會執行__init__()方法
- 如果沒有傳回第一個參數的類別實例（傳回別的實例或None），則不會執行__init__()方法（即使有定義）。

```
__new__  
__init__  
True  
-----  
__new__
```

```
class Some:  
    def __new__(cls, isClsInstance):  
        print('__new__')  
        if isClsInstance:  
            return object.__new__(cls)  
        else:  
            return None  
    def __init__(self, isClsInstance):  
        print('__init__')  
        print(isClsInstance)  
Some(True)  
print('-----')  
Some(False)
```

__new__()之範例

藉由定義__new__()方法，可以決定如何建構物件與初始物件

xlogging.py

```
1 class Logger:
2     __loggers = {}
3     def __new__(cls, name):
4         if name not in cls.__loggers:
5             logger = object.__new__(cls)
6             cls.__loggers[name] = logger
7             return logger
8         return cls.__loggers[name]
9
10    def __init__(self, name):
11        if 'name' not in vars(self):
12            self.name = name
13
14    def log(self, message):
15        print('{name}: {message}'.format(name = self.name, message = message))
```

```
import xlogging

logger1 = xlogging.Logger('xlogging')
logger1.log('一些日誌訊息....')

logger2 = xlogging.Logger('xlogging')
logger2.log('另外一些日誌訊息....')

logger3 = xlogging.Logger('xlog')
logger3.log('再來一些日誌訊息....')

print(logger1 is logger2)
print(logger1 is logger3)

xlogging: 一些日誌訊息....
xlogging: 另外一些日誌訊息....
xlog: 再來一些日誌訊息....
True
False
```

練習

- 改寫 bankclass.py，使得同樣號碼的帳戶會指向同一個物件

bankClass.py

```
1 class Account:
2     def __init__(self, name, number, balance):
3         self.name = name
4         self.number = number
5         self.balance = balance
6
7     def deposit(self, amount):
8         if amount <= 0:
9             print('存款金額不得為負')
10        else:
11            self.balance += amount
12
13    def withdraw(self, amount):
14        if amount > self.balance:
15            print('餘額不足')
16        else:
17            self.balance -= amount
18    def __str__(self):
19        return "Account('{0}', '{1}', '{2}')" .format(
20            self.name, self.number, self.balance
21        )
```