**Python程式設計：物件導向程式設計**

**目錄**

[1. 基本概念 1](#_Toc207035845)

[2. 練習 1](#_Toc207035846)

[2.1 Problem: 1](#_Toc207035847)

[2.2 Problem: 1](#_Toc207035848)

[2.3 Problem: 2](#_Toc207035849)

[2.4 Problem: 2](#_Toc207035850)

[2.5 Problem: 2](#_Toc207035851)

[2.6 Problem: 3](#_Toc207035852)

[2.7 Problem: 3](#_Toc207035853)

# 基本概念：物件導向程式設計是什麼

想像一下，我們在寫程式時，如果能用一種更貼近真實世界的方式來思考和組織程式碼，是不是會更直覺？物件導向程式設計 (OOP) 就是這樣一種方法。

核心概念有兩個：類別 (Class) 和 物件 (Object)。

* **類別 (Class)**：可以把它想像成一張「藍圖」或「設計圖」。它定義了一群東西共同擁有的**屬性 (Attributes)** 和**方法 (Methods)**。例如，我們可以設計一張「銀行帳戶」的藍圖，定義它應該有「戶名」、「餘額」這些屬性，以及「存款」、「提款」這些可以操作的方法。
* **物件 (Object)**：是根據「藍圖」(類別) 實際創造出來的「實例 (Instance)」。例如，用「銀行帳戶」這張藍圖，我們可以為 John 和 Mary 各自建立一個真實的帳戶物件。John 的帳戶和 Mary 的帳戶都擁有戶名和餘額，也都能存款和提款，但他們各自的戶名和餘額是獨立的。

為什麼要用類別？

* 程式碼重複使用：定義好一個類別，就能快速建立出許多功能相同但資料獨立的物件。
* 程式結構清晰：將相關的資料和功能綁在一起，讓程式碼更容易理解和維護。

**class** **Banks():**

bankname **=** "台南銀行"

**def** f**(**self**):**

**return** "甲方是我拔拔"

## 範例：定義一個簡單的類別

class Banks():

    bankname = "台南銀行"

    def f(self):

        return "甲方是我拔拔"

userbank = Banks() # 根據 Banks 類別建立一個 userbank 物件

# 呼叫物件的屬性和方法

print("目前服務的銀行是:", userbank.bankname)

print("銀行的服務理念是:", userbank.f())

# 物件初始化：\_\_init\_\_() 建構方法

我們在建立物件時，通常希望能夠一開始就給它一些專屬的初始資料，例如開戶時就要設定戶名和初始金額。這時候就要用到一個特殊的方法叫做 \_\_init\_\_()，它被稱為建構方法 (Constructor)。這個方法會在建立物件的當下「自動被呼叫」，讓我們可以傳入參數來設定物件的初始屬性。

## 範例：建立物件時設定初始值

class Banks():

    bankname = "台南銀行"

    def \_\_init\_\_(self, name, money):

        self.name = name

        self.money = money

    def get\_balance(self): # 定義一個取得餘額的方法

        return self.money

# 建立 hungbank 物件，並傳入初始值

hungbank = Banks('JF', 1000)

print(hungbank.name, "目前的存款是:", hungbank.get\_balance())

## 範例(練習)：增加【存款函數】跟【提款函數】

class Banks():

    bankname = "台南銀行"

    def \_\_init\_\_(self, name, money):

        self.name = name

        self.money = money

    def get\_money(self): # 定義一個取得餘額的方法

        return self.money

    def save\_money(self, money):

        self.money += money

        print(f"存錢{money}完成")

    def withdraw\_money(self, money):

        self.money -= money

        print(f"提款{money}完成")

# 建立 hungbank 物件，並傳入初始值

hungbank = Banks('JF', 1000)

print(hungbank.name, "目前的存款是:", hungbank.get\_money())

hungbank.save\_money(200)

hungbank.withdraw\_money(100)

print(hungbank.name, "目前的存款是:", hungbank.get\_money())

# 封裝 (Encapsulation)

在物件導向程式設計 (Object-Oriented Programming) 中，封裝是將物件的資料（屬性）和操作資料的方法（函式）打包在一起，並隱藏物件的內部細節，只提供一個公開的介面讓外界使用。

想像一下你家裡的微波爐

你怎麼使用它？你只需要操作外面的按鈕（設定時間、火力）然後按「開始」。你不需要知道微波爐內部的磁控管、變壓器是怎麼運作的。

為什麼這樣設計？

* 安全： 製造商把高壓電、微波等危險的零件「封裝」在金屬外殼裡，保護你不被電到或被微波傷害。
* 簡單： 你只需要學習使用那幾個按鈕（公開的介面），而不用成為電學專家。

## 公有 (Public)

就像微波爐的按鈕，是設計給外部使用的。在 Python 中，所有你正常定義的屬性和方法，預設都是公有的。

* **公有屬性 (Public attribute):** 可以在類別外部被自由讀取和修改。
* **公有方法 (Public method):** 可以在類別外部被自由呼叫。

class Banks():

    bankname = "台南銀行"

    def \_\_init\_\_(self, name, money):

        self.name = name

        self.money = money

    def get\_money(self): # 定義一個取得餘額的方法

        return self.money

    def save\_money(self, money):

        self.money += money

        print(f"存錢{money}完成")

    def withdraw\_money(self, money):

        self.money -= money

        print(f"提款{money}完成")

# 建立 hungbank 物件，並傳入初始值

hungbank = Banks('JF', 1000)

# hungbank.withdraw\_money(10000)

hungbank.money = -10000 # 餘額變成負數，這不合理！

print(hungbank.name, "目前的存款是:", hungbank.get\_money()) # 資料變得不安全

## 私有 (Private)

就像微波爐內部受保護的零件，不應該被外部直接碰觸。它們是物件內部自己運作時才需要用到的。

* 私有屬性 (Private attribute): 只能在類別內部存取，外部無法直接讀取或修改。
* 私有方法 (Private method): 只能在類別內部呼叫，外部無法直接呼叫。

解決方案：透過將 money設為私有，我們強迫使用者必須透過我們設計好的方法（如 save、withdraw）來操作餘額，我們就可以在這些方法中加入檢查機制，確保資料的安全性與正確性。

## 範例：money改私有屬性

【程式檔】私有方法\_增加匯率.py

## 範例：getter與setter

以分數存取為例，將score設定私有屬性，然後藉由getter與setter去操作

class Score():

    def \_\_init\_\_(self, score):

        self.\_\_score = score

    def get\_score(self): # setter

        return self.\_\_score

    def set\_score(self, score): # getter

        self.\_\_score = score

stu = Score(0)

print(stu.get\_score())  # 呼叫 getter

stu.set\_score(80)       # 呼叫 setter

print(stu.get\_score())  # 再次呼叫 getter 確認

## 裝飾器 (Decorator)：@property

這是一種比較偏python風格的東西

裝飾器 (Decorator) 可以讓你把類別中的「方法 (Method)」包裝成看起來像「屬性 (Attribute)」一樣，讓程式碼更簡潔、更直觀，同時保有方法的安全性。

在過往getter與setter方法中，可以注意到，單單針對score：卻要寫成 stu.get\_score() 和 stu.set\_score(95) 這種函式呼叫的形式，感覺有點囉嗦。如果能像操作普通屬性一樣寫成 stu.score 和 stu.score = 95，那該有多好？

class Score():

    def \_\_init\_\_(self, score):

        self.\_\_score = score

    @property # 將 score() 方法變成一個屬性

    def get\_score(self): # setter

        return self.\_\_score

    @get\_score.setter # 將 setter方法便屬性

    def set\_score(self, score): # getter

        self.\_\_score = score

stu = Score(0)

print(stu.get\_score)  # 呼叫 getter

stu.set\_score=80       # 呼叫 setter

print(stu.get\_score)  # 再次呼叫 getter 確認

# 實例方法 (Instance Method)

這是我們到目前為止最常用的方法類型。它的特點是，第一個參數永遠是 self，代表「這個物件實例本身」。

用來存取或修改物件自己的屬性 (Attribute)。每個物件的資料都是獨立的。例如，你的銀行帳戶餘額和我的銀行帳戶餘額是分開的。

呼叫方式：必須先建立一個物件 (實例化)，然後才能透過該物件來呼叫。

class Account:

    def \_\_init\_\_(self, name, balance):

        self.name = name       # 這是實例屬性

        self.balance = balance # 這是實例屬性

    # 這是一個實例方法，因為它有 self

    def show\_balance(self):

        print(f"{self.name} 的餘額是 {self.balance} 元")

# 必須先建立物件，再由物件呼叫實例方法

my\_acc = Account("小明", 5000)

her\_acc = Account("小華", 8000)

my\_acc.show\_balance()  # 顯示 "小明 的餘額是 5000 元"

her\_acc.show\_balance() # 顯示 "小華 的餘額是 8000 元"

# 類別方法 (Class Method)

有時候，某些資料是屬於整個類別共享的，而不是單一物件。例如，計算總共建立了多少個物件。這時就需要類別方法。

* 方法上面必須加上裝飾器 @classmethod。
* 第一個參數習慣上命名為 cls (代表 Class)，而不是 self。cls 指的是「這個類別本身」。

呼叫方式：可以直接透過類別名稱來呼叫，不需要建立物件。

class Counter:

    # 這是類別屬性，所有 Counter 物件都共享這一個 counter

    counter = 0

    def \_\_init\_\_(self):

        Counter.counter += 1 # 每當一個新物件被建立時，就更新「類別」的 counter

    # 這是類別方法，方法上面必須加上裝飾器 @classmethod

    @classmethod

    def show\_counter(cls): # 第一個參數習慣上命名為 cls (代表 Class)，而不是 self。cls 指的是「這個類別本身」。

        print(f"counter = {Counter.counter}")

# 不需要建立物件，就可以直接用類別名稱呼叫

Counter.show\_counter()

one = Counter()

two = Counter()

three = Counter()

Counter.show\_counter()

# 靜態方法 (Static Method)。

靜態方法是最簡單的一種。它基本上就是一個放在類別裡面的普通函式，只是為了組織程式碼或邏輯關聯性而這麼做。

* 方法上面必須加上裝飾器 @staticmethod。
* 沒有 self 或 cls 參數。它無法存取任何物件或類別的狀態。

當某個功能與類別有關，但又完全獨立，不需要用到類別或物件的任何資料時使用。可以把它想成是一個放在類別命名空間下的工具函式。

呼叫方式：可以直接透過類別名稱來呼叫。

class Pizza:

    # 這是靜態方法

    @staticmethod

    def demo():

        print("I like Pizza")

# 不需要建立 Pizza 物件，直接用類別名稱呼叫

Pizza.demo()

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 類型 | 裝飾器 | 第一個參數 | 功能 | 何時使用 |
| 實例方法 | (無) | self (物件實例) | 操作個別物件的資料 | 最常用，處理每個物件獨有的屬性。 |
| 類別方法 | @classmethod | cls (類別本身) | 操作整個類別共享的資料 | 需要讀取或修改所有物件共享的狀態時。 |
| 靜態方法 | @staticmethod | (無) | 獨立功能，像個工具函式 | 功能與類別相關，但不需要存取類別或物件的任何資料。 |

# 練習

## Problem:

|  |  |
| --- | --- |
| Problem Description: | |
| Input: | Output: |
| Sample Input:  13 | Sample Output: |
| Sample Input: | Sample Outpu: |
| Answer: | |

## Problem:

|  |  |
| --- | --- |
| Problem Description: | |
| Input: | Output: |
| Sample Input:  13 | Sample Output: |
| Sample Input: | Sample Outpu: |
| Answer: | |

## Problem:

|  |  |
| --- | --- |
| Problem Description: | |
| Input: | Output: |
| Sample Input:  13 | Sample Output: |
| Sample Input: | Sample Outpu: |
| Answer: | |

## Problem:

|  |  |
| --- | --- |
| Problem Description: | |
| Input: | Output: |
| Sample Input:  13 | Sample Output: |
| Sample Input: | Sample Outpu: |
| Answer: | |

## Problem:

|  |  |
| --- | --- |
| Problem Description: | |
| Input: | Output: |
| Sample Input:  13 | Sample Output: |
| Sample Input: | Sample Outpu: |
| Answer: | |

## Problem:

|  |  |
| --- | --- |
| Problem Description: | |
| Input: | Output: |
| Sample Input:  13 | Sample Output: |
| Sample Input: | Sample Outpu: |
| Answer: | |

## Problem:

|  |  |
| --- | --- |
| Problem Description: | |
| Input: | Output: |
| Sample Input:  13 | Sample Output: |
| Sample Input: | Sample Outpu: |
| Answer: | |