Python 程式設計：物件導向程式設計

目錄

1. 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"
```

1.1 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())
```

2. 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。

2.1 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())
```

2.2 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())
```

3. 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。

3.1 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
```

3.2 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。

3.3 範例：money 改私有屬性

【程式檔】私有方法_增加匯率.py

3.4 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
```

3.5 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
```

4. 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"
```

5. 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()
```

6. 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。

7. 練習

7.1 Problem: 安全的銀行帳戶

Problem Description:

位於台南市東區的「台南銀行」正在開發新一代的網路銀行系統。他們需要一個 Account 類別來表示客戶帳戶。一個帳戶有戶名 (name)、餘額 (balance) 以及一個初始設定後就不能再被外部更改的帳戶號碼 (account_number)。為了安全性，餘額 balance 絕對不能從外部被直接設定為任意數字，尤其是負數。所有餘額的變動都必須透過 deposit (存款) 和 withdraw (提款) 方法來進行。存款金額必須是正數，而提款金額也必須是正數，且不能超過當前餘額。任何無效的操作都應該印出錯誤訊息。

請你設計這個 Account 類別，確保其資料的安全性與完整性。

Input:	Output:
輸入共四行。 第一行是三個由空白隔開的字串：初始戶名 name、帳戶號碼 account_number、初始餘額 balance。 第二行是一個字串 deposit 與一個整數，代表存款操作與金額。 第三行是一個字串 withdraw 與一個整數，代表提款操作與金額。 第四行是一個字串 withdraw 與一個整數，代表第二次提款操作與金額。	針對每一次有效操作，不需要印出訊息。 針對無效操作（存款或提款金額為負、提款超過餘額），需印出一行錯誤訊息 Invalid operation.。 最後，印出帳戶的最終狀態，格式為 Account [帳戶號碼] ([戶名]) has $[最終餘額].。
Sample Input: Che-Wei_Chen A123456789 5000 deposit 3000 withdraw 1500 withdraw 8000	Sample Output: Invalid operation. Account A123456789 (Che-Wei_Chen) has $6500.
Sample Input: Guest B987654321 1000 deposit -500 withdraw 200 withdraw 800	Sample Outpu: Invalid operation. Account B987654321 (Guest) has $0.
Answer: 安全的銀行帳戶.py	

7.2 Problem: 安全的銀行帳戶

Problem Description:

你正在開發一款角色扮演遊戲 (RPG)。請建立一個 Character 類別。角色擁有名字 (name)、等級 (level)、生命值 (hp) 和經驗值 (exp)。

這些屬性有以下規則：

- hp 是私有的，且其值永遠不能超過角色的最大生命值 max_hp。最大生命值 max_hp 的計算公式為 level * 10。
- level 和 exp 也是私有的。
- 等級 level 初始為 1。
- 提供一個公有方法 gain_exp(points) 來增加經驗值。每當經驗值累積到 level * 100 時，角色就會升級，level 增加 1，同時 hp 會自動補滿到新的 max_hp。
- hp 可以被外部像屬性一樣讀取和設定，但設定時必須遵守規則 1。

請使用 @property 裝飾器來實作 hp 的存取。

Input:	Output:
輸入有多行。 第一行是角色的名字。 接下來每一行都是一個操作，格式為 action value。 action 可能是 set_hp (試圖設定 hp) 或 gain_exp (獲得經驗值)。 輸入以 END 結束。	在所有操作結束後，輸出角色的最終狀態，格式為： Name: [名字] Level: [等級] HP: [目前 HP]/[最大 HP] EXP: [目前經驗值]
Sample Input: Hero gain_exp 150 set_hp 5 gain_exp 200 END	Sample Output: Name: Hero Level: 3 HP: 30/30 EXP: 50
Sample Input: Mage gain_exp 50 set_hp 100 gain_exp 50 set_hp 5 END	Sample Outpu: Name: Mage Level: 2 HP: 5/20 EXP: 0

Answer:

遊戲角色控制器.py

7.3 Problem: 產品庫存管理 (Product Inventory)

Problem Description:

一間線上商店需要一個 Product 類別來管理庫存。每個產品都有品名 name、價格 price 和庫存數量 quantity。

規則如下：

- price 和 quantity 必須是私有屬性。
- price 和 quantity 必須透過 getter 和 setter 方法來存取。

<table>
<tr><td colspan="2">

set_price 方法必須確保價格不能為負數。
set_quantity 方法必須確保庫存數量不能為負數。
如果試圖設定無效的值，屬性的值應保持不變，並印出錯誤訊息 Invalid value.。
提供一個 get_total_value 方法，計算並返回 price * quantity 的總價值。

</td></tr>
<tr><td>Input:
輸入共四行。
第一行是三個由空白隔開的值：初始品名、價格、數量。
第二行是 set_price 與一個新價格。
第三行是 set_quantity 與一個新數量。
第四行是 set_quantity 與另一個新數量。</td><td>Output:
針對無效的設定操作，印出 Invalid value.。
最後，印出產品的最終狀態，格式為：
Product: [品名]
Price: $[價格]
Quantity: [數量]
Total Value: $[總價值]</td></tr>
<tr><td>Sample Input:
Invalid value.
Product: Laptop
Price: $1150
Quantity: 8
Total Value: $9200</td><td>Sample Output:
Laptop 1200 10
set_price 1150
set_quantity -5
set_quantity 8</td></tr>
<tr><td>Sample Input:
Invalid value.
Product: Keyboard
Price: $75
Quantity: 30
Total Value: $2250</td><td>Sample Outpu:
Keyboard 75 20
set_price -10
set_quantity 15
set_quantity 30</td></tr>
<tr><td colspan="2">Answer:
產品庫存管理.py</td></tr>
</table>

8. 繼承(Inheritance)

想像一下，你已經寫好一個很棒的 class 汽車，它有「加速」、「煞車」等方法。但現在你想新增一個 class 跑車。這時候，你就會有幾個選擇：

- 修改 class 汽車：在裡面加入跑車才有的功能。但這樣會讓原來的 class 汽車越來越複雜，而且如果以後要新增 class 卡車 呢？
- 複製貼上：把 class 汽車的程式碼全部複製一份，然後改名成 class 跑車再新增功能。但這樣未來如果要修改一個 bug，你就得改兩個地方，非常難以維護。

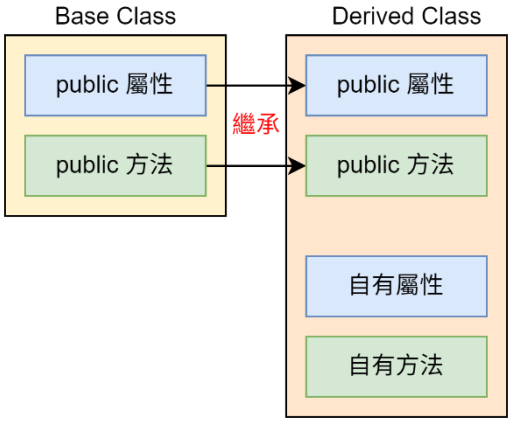
繼承就是為了解決這個問題而生的。它讓你可以這麼做：「讓新的 class 跑車，直接『繼承』class 汽車的所有功能，然後我們再專心在 class 跑車 裡加入它獨有的新功能（例如：開啟氮氣加速）。」

在繼承的世界裡，我們有兩種主要的角色：

- 父類別 (Parent Class)：也被稱為 基礎類別 (Base Class)。它是被繼承的那個類別（例如：汽車）。
- 子類別 (Child Class)：也被稱為 衍生類別 (Derived Class)。它是繼承者，會擁有父類別所有公開的屬性與方法

（例如：跑車）。

```
class 父類別名稱:
    # ... 父類別的程式碼 ...

class 子類別名稱(父類別名稱):
    # ... 子類別的程式碼 ...
```



8.1 範例：Derived Class 繼承 Base Class 基本用法

這個範例展示了最純粹的繼承。Son 類別本身是空的 (pass)，但它繼承了 Father，因此獲得了 Father 的所有能力。

```
class Father():
    def hometown(self):
        print("我住台南")

class Son(Father):
    pass

ta1 = Father()
ta2 = Son()
ta1.hometown()
ta2.hometown()
```

8.2 範例：方法覆寫 (Method Overriding)

如果子類別不滿意父類別某個方法的實現，可以在自己內部重新定義一個同名的方法，這會覆蓋掉父類別的版本。這個行為稱為 覆寫 (Overriding)。

```
# 定義父類別 Person
class Person():
    def job(self):
        print("我是一般人")
```

```
# 定義子類別 LawyerPerson，繼承 Person
class LawyerPerson(Person):
    # 覆寫了父類別的 job() 方法
    def job(self):
        print("我是律師")

# 實例化
hung = Person()
ivan = LawyerPerson()

hung.job()  # 執行 Person 類別的 job()
ivan.job()  # 執行 LawyerPerson 類別覆寫後的 job()
```

8.3 使用 super() 呼叫並擴充父類別方法

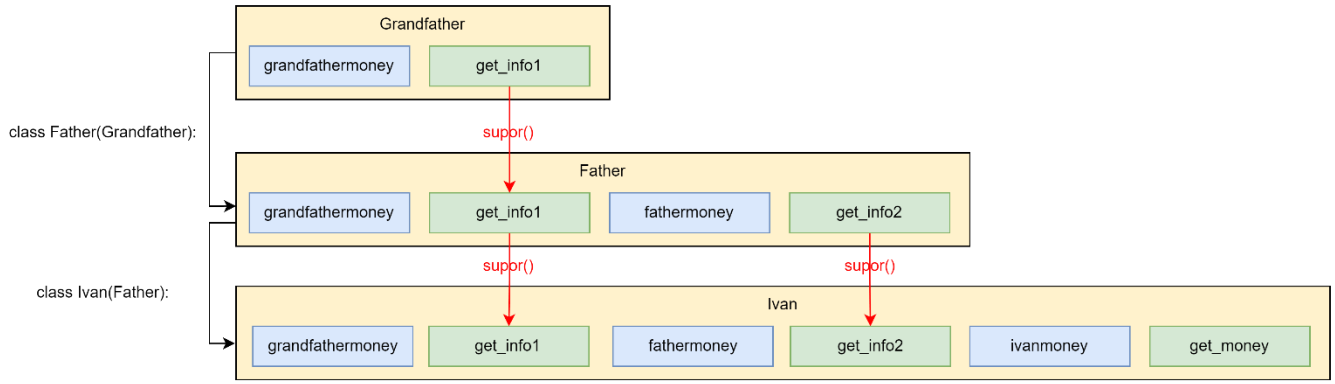
覆寫是「完全取代」，但很多時候我們只想在父類別的基礎上「增加」新功能。這時 super() 就非常關鍵了。super() 是一個特殊函數，可以幫助我們在子類別中呼叫到父類別的方法。最常見的應用是在 __init__() 建構子中。

```
class Animal():
    def __init__(self, animal_name, animal_age):
        print("父類別 Animal 的 __init__ 被執行了")
        self.name = animal_name
        self.age = animal_age

    def run(self):
        print(self.name, "is running")


class Dogs(Animal): # 定義子類別 Dogs，繼承 Animal
    def __init__(self, dog_name, dog_age, dog_title):
        print("子類別 Dogs 的 __init__ 被執行了")
        # 使用 super() 呼叫父類別 Animal 的 __init__ 方法
        # 將 name 和 age 的初始化工作交給父類別完成
        super().__init__(dog_name, dog_age)

        self.title = dog_title # 子類別只需處理自己獨有的屬性

mydog = Dogs('Lily', 6, 'My Pet Lily')
print(f"名字: {mydog.name}, 年齡: {mydog.age}") # mydog.name 和 mydog.age 是由父類別的
__init__ 初始化的
```

```
print(f"稱號: {mydog.title}") # mydog.title 是由子類別的 __init__ 初始化的
```

8.4 範例：多層繼承 (Multi-level Inheritance)

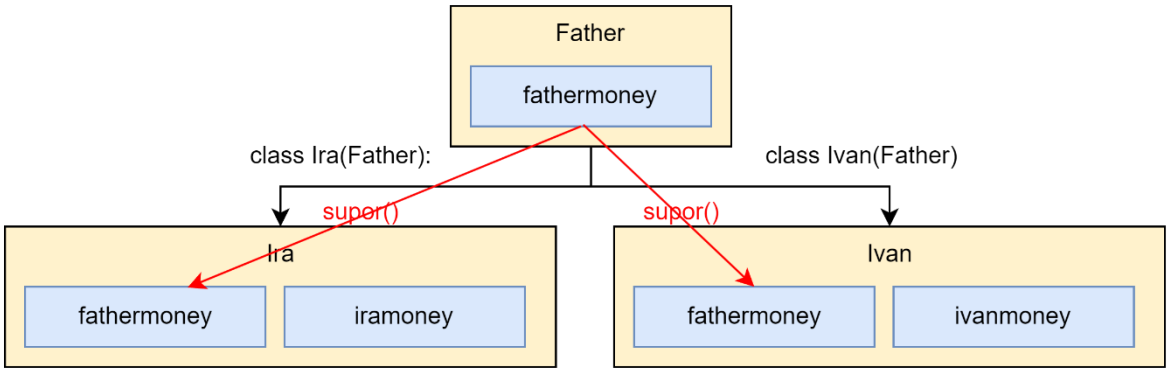
繼承關係可以形成一個鏈條：Ivan 繼承 Father，而 Father 又繼承 Grandfather。在這種結構下，最底層的 Ivan 會擁有所有祖先的特性。




```
class Grandfather():
    def __init__(self):
        self.grandfathermoney = 10000

    def get_info1(self):
        print("Grandfather's information")

class Father(Grandfather):
    def __init__(self):
        super().__init__()
        self.fathermoney = 8000

    def get_info2(self):
        print("Father's information")

class Ivan(Father):
    def __init__(self):
        super().__init__()
        self.ivanmoney = 5000

    def get_money(self):
        print("Ivan 的資產:", self.ivanmoney)
        print("父親的資產:", self.fathermoney) # 繼承自 Father
        print("祖父的資產:", self.grandfathermoney) # 繼承自 Grandfather

ivan = Ivan()
```

```
ivan.get_info1()  # 來自 Grandfather 的方法
ivan.get_info2()  # 來自 Father 的方法

# 呼叫自身的方法，並存取所有繼承來的屬性
ivan.get_money()
```

8.5 範例：兄弟類別




```
class Father():
    """定義父親的資產"""
    def __init__(self):
        self.fathermoney = 10000

class Ira(Father):   # 父類別是 Father
    """定義 Ira 的資產"""
    def __init__(self):
        super().__init__()
        self.iramoney = 8000


class Ivan(Father):  # 父類別是 Father
    """定義 Ivan 的資產"""
    def __init__(self):
        self.ivanmoney = 3000
        super().__init__()

    def get_money(self): # 取得資產明細
        print("Ivan 資產: ", self.ivanmoney,
              "\n 父親資產: ", self.fathermoney,
              "\nIra 資產: ", Ira().iramoney) # 注意寫法

# 建立物件實例與測試
```

```
ivan = Ivan()
ivan.get_money()
```

整理：
「繼承」這個行為，在您寫下 class 子類別名稱(父類別名稱): 的那一刻就已經完成了。
當我們寫完 class 子類別名稱(父類別名稱):，子類別就立刻、自動地繼承了父類別所有公開的 (public) 屬性 (attributes) 和方法 (methods)。**兩者是同時繼承的，並不需要分開處理。**

那 super()是幹嘛的?
super() 的功能不是用來「繼承」方法的。super() 的用途是：當你在子類別中「覆寫 (override)」了父類別的某個方法時，如果你還想在新的方法中呼叫「父類別原本的功能」，就要使用 super()。

8.6 情境 A：單純使用繼承的方法 (不需 super())

```
class Father:
    def work(self):
        print("父親在傳統產業工作")

class Son(Father):
    pass

Tom = Son()
Tom.work() # 直接使用繼承來的方法
```

8.7 情境 B：覆寫 (Override) 父類別的方法 (也不需 super())

```
class Father:
    def work(self):
        print("父親在傳統產業工作")

class Son(Father):
    def work(self): # 重新定義了 work()，完全覆蓋父類別的版本
        print("兒子在科技業工作")
Tom = Son()
Tom.work() # 直接使用繼承來的方法
```

8.8 情境 C：在覆寫的基礎上，使用 super() 擴充功能 (這才是 super() 的用武之地)

```
class Father:
    def work(self):
        print("父親在傳統產業工作")

class Son(Father):
    def work(self):
```

```
        # 我希望先執行父類別的 work()，再執行我自己的
        super().work() # 用 super() 呼叫父類別被覆寫掉的 work() 方法
        print("兒子也在科技業找到了新工作") # 加上自己的新功能
```

這個情境 C，尤其是在 __init__() 建構子中，是最常見也最標準的 super() 用法。子類別的 __init__ 需要先用 super().__init__() 去執行父類別的初始化流程，以確保父類別的屬性被正確建立，然後再加入子類別自己獨有的屬性。

9. 練習

9.1 Problem: 職業的繼承

Problem Description:

在一個角色扮演遊戲中，所有角色都源自於一個基礎的 Adventurer (冒險者) 類別。Adventurer 類別在創建時需要提供名字 name 和生命值 hp。它有一個 show_status() 方法，可以顯示角色的基本資訊。

現在，請你設計一個 Knight (騎士) 類別，它繼承自 Adventurer。Knight 除了擁有冒險者的所有特性外，還額外擁有一個 armor (盔甲值) 屬性。

你的任務是：

1. 建立 Adventurer 類別，其 __init__ 方法接收 name 和 hp，並實作 show_status() 方法。
2. 建立 Knight 類別，繼承 Adventurer。
3. Knight 的 __init__ 方法需要接收 name, hp, 和 armor。請務必使用 super() 來完成 name 和 hp 的初始化。
4. Knight 類別需要**覆寫 (override)** show_status() 方法，使其在顯示基本資訊的基礎上，額外顯示盔甲值。
5. Knight 類別需要有一個自己獨有的新方法 defend()，呼叫此方法會印出特定訊息。

Input:	Output:
輸入共有兩行。 第一行包含一個字串 name 和一個整數 hp，代表一位騎士的姓名和生命值，兩者以空格分隔。 第二行只包含一個整數 armor，代表這位騎士的盔甲值。	請根據騎士的資訊，實例化一個 Knight 物件，並依序呼叫 show_status() 和 defend() 方法。 show_status() 的輸出格式為 Name: [name], HP: [hp], Armor: [armor]。 defend() 的輸出格式為 [name] is defending with armor [armor]!。
Sample Input: Arthur 150 50	Sample Output: Name: Arthur, HP: 150, Armor: 50 Arthur is defending with armor 50!
Sample Input: Lancelot 120 75	Sample Outpu: Name: Lancelot, HP: 120, Armor: 75 Lancelot is defending with armor 75!
Answer:	

9.2 Problem: 校園成員名冊

Problem Description:

一所大學想要建立一個系統來管理所有成員的資料。所有成員 (Member) 都有 name (姓名) 和 member_id (成員編號)。這個系統中有兩種主要的**兄弟類別**：Student (學生) 和 Professor (教授)，它們都繼承自 Member。

- Student 除了基本資料外，還有一個 major (主修) 屬性。
- Professor 除了基本資料外，還有一個 department (系所) 屬性。

請你實作這三個類別，並確保 Student 和 Professor 的 __init__ 方法都使用 super() 來初始化繼承來的屬性。程式需要根據輸入的類型，建立對應的物件 (Student 或 Professor)，並呼叫 display() 方法來印出該成員的完整資訊。

Input:	Output:
輸入共有兩行。 第一行是一個字元，S 代表學生，P 代表教授。 第二行包含一個字串 (姓名)、一個整數 (編號) 和一個字串 (主修或系所)，均以空格分隔。	根據輸入的類型，印出一行格式化的成員資訊。 如果是學生，格式為 Student Record: [name]([member_id]) - Major: [major] 如果是教授，格式為 Professor Record: [name]([member_id]) - Department: [department]
Sample Input: S David 110062001 ComputerScience	Sample Output: Student Record: David(110062001) - Major: ComputerScience
Sample Input: P Dr.Chen 70503022 ElectricalEngineering	Sample Outpu: Professor Record: Dr.Chen(70503022) - Department: ElectricalEngineering
Answer:	

9.3 Problem: 交通工具的演化

Problem Description:

這個問題將測試你對**多層繼承 (Multi-level Inheritance)** 的理解。 你需要建立一個三層的繼承結構：Vehicle -> Car -> ElectricCar。

1. **Vehicle (交通工具) 類別**：
   - __init__ 方法接收一個 max_speed (最高時速) 屬性。
2. **Car (汽車) 類別**：
   - 繼承自 Vehicle。
   - __init__ 方法接收 max_speed 和 num_doors (車門數)。它需要使用 super() 呼叫 Vehicle 的 __init__。
3. **ElectricCar (電動車) 類別**：
   - 繼承自 Car。
   - __init__ 方法接收 max_speed, num_doors, 和 battery_capacity (電池容量, kWh)。它需要使用 super() 呼叫 Car 的 __init__。
   - 擁有一個 display_specs() 方法，需要印出從各層繼承來的所有屬性。

你的任務是讀取一行輸入來建立一個 ElectricCar 物件，並呼叫其 display_specs() 方法。	
Input: 輸入只有一行，包含三個以空格分隔的整數，依序代表 max_speed, num_doors, 和 battery_capacity。	Output: 輸出三行，分別顯示電動車的三個屬性，格式如下： Max Speed: [value] km/h Number of Doors: [value] Battery Capacity: [value] kWh
Sample Input: 250 4 100	Sample Output: Max Speed: 250 km/h Number of Doors: 4 Battery Capacity: 100 kWh
Sample Input: 180 2 75	Sample Outpu: Max Speed: 180 km/h Number of Doors: 2 Battery Capacity: 75 kWh
Answer:	

10. 多型 (Polymorphism)

多型 (Polymorphism) 的字面意思是「多種型態」。在程式設計中，它指的是不同的物件，可以對同一個指令（方法呼叫）做出各自不同的回應。這讓我們的程式碼可以寫得更通用、更有彈性。我們可以寫一個函數，它不需要知道傳進來的物件「確切」是什麼類別，只需要知道它「能夠」執行某個方法即可。

10.1 範例：基本多型範例

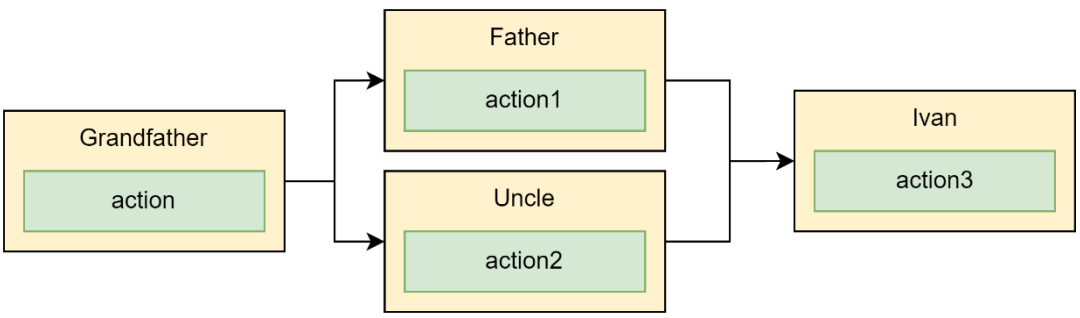
```
class Animals():
    """ Animals 類別，這是基底類別 """
    def __init__(self, animal_name):
        self.name = animal_name
    def which(self):
        return 'My pet ' + self.name.title()
    def action(self):
        print("sleeping")

class Dogs(Animals):
    """ Dogs 類別，這是 Animal 的衍生類別 """
    def __init__(self, dog_name, dog_title):
        super().__init__(dog_name)
        self.title = dog_title
    def action(self): # 覆寫了父類別的 action 方法
        print(self.name.title(), "is running in the street")
```

```
class Monkeys():
    """ 一個與 Animal 無關的獨立類別 """
    def __init__(self, monkey_name):
        self.name = monkey_name
    def which(self):
        return self.name.title()
    def action(self):
        print(self.name.title(), "is running in the forest")

# 這是一個通用的函數，它只在乎傳進來的 obj 物件有沒有 which() 和 action() 方法
def doit(obj):
    print(obj.which())
    obj.action()

# 實例化三個不同類別的物件
my_cat = Animals('lucy')
my_dog = Dogs('gigi', 'my pet')
my_monkey = Monkeys('taylor')

# 將不同物件傳入同一個 doit() 函數
print("執行 my_cat:")
doit(my_cat)
print("\n 執行 my_dog:")
doit(my_dog)
print("\n 執行 my_monkey:")
doit(my_monkey)
```

10.2 多重繼承 (Multiple Inheritance) - 擁有多位父親

Python 允許一個子類別同時繼承多個父類別，這稱為多重繼承。子類別會繼承**所有**父類別的屬性與方法。

```
class ChildClass(ParentClass1, ParentClass2, ...):
    pass
```



```
# ch12_19.py

class Grandfather():
    """ 定義祖父 """
    def __init__(self):
        print("Grandfather's __init__")
    def action(self):
        print("Grandfather")

class Father(Grandfather):
    """ 定義父親，繼承祖父 """
    def __init__(self):
        print("Father's __init__")
    def action1(self):
        print("Father")

class Uncle(Grandfather):
    """ 定義叔父，繼承祖父 """
    def __init__(self):
        print("Uncle's __init__")
    def action2(self):
        print("Uncle")

# Ivan 同時繼承 Father 和 Uncle
class Ivan(Father, Uncle):
    """ 定義 Ivan """
    def __init__(self):
        print("Ivan's __init__")
    def action3(self):
        print("Ivan")

# 實例化 Ivan
ivan = Ivan()

# 呼叫來自不同父類別與祖父類別的方法
ivan.action3() # 呼叫自己的方法
ivan.action1() # 繼承自 Father 的方法
ivan.action2() # 繼承自 Uncle 的方法
ivan.action()  # 繼承自 Grandfather 的方法
```

10.3 範例：鑽石繼承

但是，多重繼承會帶來一個複雜的問題，尤其是在 super() 的使用上。在單純繼承中，super() 會呼叫父類別。但在多重繼承中，它會呼叫誰？

```
class A:
    def __init__(self):
        print("Class A")
class B:
    def __init__(self):
        print("Class B")
class C(A, B): # C 繼承 A 和 B
    def __init__(self):
        super().__init__()
        print("Class C")

x = C()
# Class A
# Class C

#  C 的 super().__init__() 只執行了 A 的 __init__，而 B 的 __init__ 被完全忽略了！這是因為 C 的 MRO 是 [C, A, B, object]，super() 呼叫完 A 之後，鏈條就斷了。
```

解法，在使用多重繼承時，為了確保所有父類別的初始化方法都能被執行，繼承鏈中的每一個類別都應該使用 super()。

```
class A:
    def __init__(self):
        super().__init__()
        print("Class A")
class B:
    def __init__(self):
        super().__init__()
        print("Class B")
class C(A, B): # C 繼承 A 和 B
    def __init__(self):
        super().__init__()
        print("Class C")

x = C()
# Class B
# Class A
```

```
# Class C
```

11. 檢查物件身份：isinstance() 與 type()

在處理繼承和多型的程式碼時，有時我們需要檢查一個物件到底是什麼來頭。Python 提供了兩個主要工具：

type(obj)：

- 回傳物件最精確、最真實的類別。
- 它非常嚴格，不會考慮繼承關係。

isinstance(obj, ClassName)

- 檢查 obj 是否為 ClassName 類別的實例，或是 ClassName 任何子類別的實例。
- 它會考慮整個繼承鏈，因此在物件導向程式設計中更常用、也更推薦。

範例：檢查物件身份.py

12. 練習

12.1 Problem:

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

12.2 Problem:

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

12.3 Problem:

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

12.4 Problem:

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

12.5 Problem:

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