Python class 說明

Python 中，\*\*類別（class）\*\*是用來建立物件的藍圖，物件則是類別的實體

1. 類別成員

1. 類別的語法範例：

class Person:

# 建構函式（Constructor）

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

self.name = name # 屬性（Attribute）

self.age = age # 屬性（Attribute）

# 方法（Method）

def greet(self):

print(f"你好，我是 {self.name}，今年 {self.age} 歲。")

2. 建構函式（\_\_init\_\_）

* 這是類別初始化時自動執行的方法。
* 用來給屬性（如 name 和 age）設定初始值。
* self 代表實體本身。

3. 屬性（Attributes）

* 屬性是物件的資料值，如上例的 name 和 age。
* 一般會在 \_\_init\_\_ 中透過 self 宣告與賦值。

4. 方法（Methods）

* 方法是定義在類別中的函式。
* 例如 greet 是讓這個人介紹自己的方法。
* 所有方法的第一個參數都是 self，代表呼叫該方法的物件本身。

1. 屬性賦值的兩種方式：建構函式 vs 物件外部直接賦值

### 方式 1：建構函式中賦值（建議）

p1 = Person("小明", 25)

print(p1.name) # 小明

**優點：**

* 更加結構化與一致。
* 建立物件時就有明確的初始狀態。
* 便於維護與閱讀。

### 方式 2：建立物件後再直接指定屬性值

p2 = Person("小美", 30)

p2.age = 35 # 修改已存在的屬性

p2.city = "台北" # 新增屬性（但不是由類別定義的）

print(p2.age) # 35

print(p2.city) # 台北

### 差異與注意事項：

* 可以動態地新增或修改屬性（Python 是動態語言）。
* 但這樣的屬性不是類別預先定義的，容易導致維護困難或拼錯名稱卻不會報錯。
* **建議做法是盡量透過建構函式與方法來管理屬性。**

1. 補充：類別的完整範例（含方法操作屬性）

class Circle:

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \*\* 2

def set\_radius(self, new\_radius):

self.radius = new\_radius

# 使用

c = Circle(5)

print(c.area()) # 印出圓面積

c.set\_radius(10)

print(c.area()) # 修改半徑後再印出新面積

1. 私有屬性（Private Attributes）

### 說明：

在 Python 中，**私有屬性並非真正封裝**，但透過命名慣例達成保護屬性不被外部直接存取。

### 使用方式：

class Person:

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

self.name = name # 公開屬性

self.\_\_age = age # 私有屬性（名稱前加雙底線）

def get\_age(self):

return self.\_\_age # 提供取值方式

def set\_age(self, new\_age):

if new\_age >= 0:

self.\_\_age = new\_age

else:

print("年齡不能為負")

# 使用

p = Person("小明", 20)

print(p.name) # 可以直接存取

# print(p.\_\_age) # ❌ AttributeError（不可直接存取）

print(p.get\_age()) # ✅ 建議透過方法取得

p.set\_age(-5) # 年齡不能為負

p.set\_age(30)

print(p.get\_age()) # 30

### 底層其實是 \_Person\_\_age，仍可強行存取（但不建議）：

print(p.\_Person\_\_age)

1. @property（屬性方法裝飾器）

### 說明：

將方法「偽裝」成屬性存取形式，使語法更自然，並可搭配 getter/setter 控制存取權限。

### 範例：

class Circle:

def \_\_init\_\_(self, radius):

self.\_radius = radius # 通常使用單底線表示保護屬性

@property

def radius(self):

return self.\_radius # getter

@radius.setter

def radius(self, value):

if value > 0:

self.\_radius = value

else:

raise ValueError("半徑必須大於 0")

@property

def area(self):

return 3.14 \* self.\_radius \*\* 2 # 只讀屬性

# 使用

c = Circle(5)

print(c.radius) # 像讀屬性一樣

print(c.area) # 像讀屬性一樣（不能寫入）

c.radius = 10 # 像設定屬性一樣

# c.area = 100 # ❌ 無 setter，會報錯

✅ **好處：**

* 控制存取邏輯（例如驗證）
* 提供「唯讀屬性」
* 語法簡潔易讀

|  |  |  |
| --- | --- | --- |
| 功能 | 說明 | 範例關鍵字 |
| 私有屬性 | 不希望外部直接操作 | self.\_\_age |
| @property | 控制屬性讀/寫方式，實現封裝 | @property、@setter |

1. 類別繼承（Inheritance）

### 功能：

讓一個類別「繼承」另一個類別的屬性與方法，避免重複撰寫相同邏輯，支援多型（Polymorphism）。

### 範例：

class Animal:

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

self.name = name

def speak(self):

print(f"{self.name} 發出聲音")

class Dog(Animal): # Dog 繼承自 Animal

def speak(self):

print(f"{self.name} 汪汪叫")

# 使用

a = Animal("動物")

d = Dog("小黑")

a.speak() # 動物 發出聲音

d.speak() # 小黑 汪汪叫（子類別可覆寫方法）

✅ **子類別可：**

* 繼承建構函式與方法
* 覆寫（Override）方法
* 使用 super() 呼叫父類別方法（例如在建構函式中）

class Cat(Animal):

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

super().\_\_init\_\_(name) # 呼叫父類別的 \_\_init\_\_

self.color = color

1. 多型（Polymorphism）

### 概念：

**多型**允許「不同類別」的物件，在相同方法名稱下有不同的實作方式。

這是物件導向程式設計（OOP）四大特性之一。

### 範例：

class Animal:

def speak(self):

print("某種動物在發出聲音")

class Dog(Animal):

def speak(self):

print("狗：汪汪！")

class Cat(Animal):

def speak(self):

print("貓：喵喵！")

def animal\_speak(animal):

animal.speak() # 不論傳入哪種動物，都呼叫對應的 speak()

# 使用

a = Animal()

d = Dog()

c = Cat()

animal\_speak(a) # 某種動物在發出聲音

animal\_speak(d) # 狗：汪汪！

animal\_speak(c) # 貓：喵喵！

✅ 重點：不同子類別可以共用一個函式介面，但表現出不同的行為。

1. 抽象類別（Abstract Class）

### 概念：

抽象類別是不能被實例化的類別，用來作為「其他類別的模板」。

通常會包含「抽象方法」，強迫子類別必須實作它們。

### 用法：

透過 abc 模組（Abstract Base Class）

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass # 抽象方法，不提供實作

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return 3.14 \* self.radius \*\* 2

class Square(Shape):

def \_\_init\_\_(self, side):

self.side = side

def area(self):

return self.side \*\* 2

# shape = Shape() # ❌ 不能實例化抽象類別

c = Circle(5)

s = Square(4)

print(c.area()) # 78.5

print(s.area()) # 16

✅ 抽象類別的用途：

* 規範子類別結構
* 定義一致的介面（如 area()）
* 避免「忘記實作」某些必要方法

1. 類別變數 vs. 實體變數

| **項目** | **實體變數** | **類別變數** |
| --- | --- | --- |
| 定義位置 | 通常在 \_\_init\_\_ 中使用 self.變數名 | 類別區塊中，直接定義 |
| 屬於 | 每個物件（實例） | 整個類別，共享 |
| 修改 | 只影響單一物件 | 所有物件共享此變數（除非被覆蓋） |

### 範例：

class Counter:

count = 0 # 類別變數（全類別共享）

def \_\_init\_\_(self):

Counter.count += 1 # 每創一個實例就加 1

self.instance\_id = Counter.count # 實體變數

# 使用

a = Counter()

b = Counter()

c = Counter()

print(Counter.count) # 3 （類別變數）

print(a.instance\_id) # 1

print(b.instance\_id) # 2

print(c.instance\_id) # 3

### 小提醒：

* 類別變數若被實體「重新指派」會變成實體專屬，類別本身的值不會變。
* 通常用於 **統計、共享設定、資源池等情境**。

1. 補充整理：

| **主題** | **重點** |
| --- | --- |
| 多型 | 同一方法名稱，不同物件可有不同實作 |
| 抽象類別 | 定義方法框架但不實作，強制子類實作 |
| 類別變數 vs. 實體變數 | 類別變數屬於整個類別，實體變數屬於個別物件 |