Python Class 類別

Class 類別



想像一下，假如我們要做一個有關食譜和食物的程式。首先，什麼是食譜呢？食譜是一個告訴我們如何做一道菜的指南，裡面會寫著需要的材料和做法。這就像是一個「類別」，它定義了一種事物的特性和行為。

現在，我們可以把不同的食物想像成「物件」，每一道菜都是一個物件。比如，我們有一個叫做「巧克力蛋糕」的食譜，這個食譜裡面會告訴我們需要的巧克力、麵粉、雞蛋等等。這個食譜就是我們的「類別」，裡面定義了製作巧克力蛋糕的方法。

當我們根據這個食譜做出一個巧克力蛋糕時，這個巧克力蛋糕就是一個「物件」，它具體實現了食譜中的方法。我們可以做出很多不同的巧克力蛋糕，但它們都是根據同一個食譜來製作的，所以它們有相同的特性和做法。

同樣地，當我們做其他的菜時，比如「水果沙拉」或「披薩」，每道菜都是一個物件，它們都有各自的特性和做法，但都是根據不同的食譜（類別）來做的。

所以，「類別」就像是一個模板或者指南，告訴我們如何創造不同的物件。這種方式讓我們可以用更有組織的方式來寫程式，就像是根據食譜來做不同的美食一樣。希望這個比喻可以幫助你們更好地理解「類別」這個概念喔！

**創建類別與實例化物件**

類別（Class）是面向物件程式設計（Object-Oriented Programming，簡稱 OOP）中的一個重要概念。在 Python 中，你可以使用 class 關鍵字來定義類別，類別用於創建物件，並定義了物件的屬性和方法。



以下是一個簡單的類別示範：

class MyClass:

x = 5

# 依據類別創建物件

p1 = MyClass()

p1.x # 5

定義一個類別 MyClass，以及如何基於這個類別創建物件 p1，並且訪問物件的屬性 x。

**建構函示 init() 與實例方法**

以下是一個簡單的類別示範：

class Dog:

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

self.name = name

self.age = age

def bark(self):

print(self.name + "正在叫！")

def describe(self):

print(self.name + "的年齡是" + str(self.age) + "歲。")

# 創建 Dog 類別的物件

dog1 = Dog("旺財", 3)

dog2 = Dog("小黑", 2)

# 使用物件的方法

dog1.bark() # 旺財正在叫！

dog2.describe() # 小黑 的年齡是 2 歲。

在這個示範中，我們定義了一個名為 Dog 的類別，它有兩個屬性 name 和 age，以及兩個方法 bark 和 describe。init 方法是一個特殊的方法，用於初始化物件的屬性。在方法中，第一個參數 self 代表物件自身。

你可以創建多個基於這個類別的物件，每個物件都有自己的屬性值和可以呼叫的方法。這種方式使你能夠更好地組織和管理你的程式碼，同時也促進了程式碼的重用性。

**類別繼承**

當你創建基礎類別（Base Class）時，你正在定義一個用於建立其他類別的模板。這個基礎類別可以包含共用的屬性和方法，供其他衍生類別（Derived Class）繼承和擴展。  
以下是一個簡單的基礎類別的示範，以及如何實例化物件：

class Animal:

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

self.name = name

def make\_sound(self):

pass # 在基礎類別中，這個方法不做任何事情

# 衍生類別 Dog，繼承自 Animal

class Dog(Animal):

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

super().\_\_init\_\_(name) # 呼叫基礎類別的初始化方法

self.breed = breed

def make\_sound(self):

return "汪汪！"

# 衍生類別 Cat，繼承自 Animal

class Cat(Animal):

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

super().\_\_init\_\_(name)

self.color = color

def make\_sound(self):

return "喵喵！"

# 創建 Dog 和 Cat 的物件

dog = Dog("旺財", "柴犬")

cat = Cat("小花", "灰色")

# 使用物件的屬性和方法

print(dog.name + "是一隻" + dog.breed + "。" + dog.make\_sound())

print(cat.name + "是一隻" + cat.color + "的貓。" + cat.make\_sound())

在這個示範中，我們創建了一個基礎類別 Animal，它具有一個共用的屬性 name 和一個未實作的方法 make\_sound。然後，我們創建了兩個衍生類別 Dog 和 Cat，它們分別繼承了 Animal 類別，並在其中實作了 make\_sound 方法。

透過使用 super() 函式，我們可以在衍生類別的 **init** 方法中呼叫基礎類別的初始化方法，以確保共用的屬性能夠正確初始化。

最後，我們創建了 Dog 和 Cat 的物件，並使用它們的屬性和方法來顯示相關訊息。

**封裝（Encapsulation）**

封裝是將資料（屬性）和操作資料的方法（方法）包裝起來的概念。這意味著外部程式碼不能直接訪問類別內部的資料，只能通過公開的方法（接口）進行訪問。這樣的設計方式可以隱藏內部的實現細節，同時確保資料的安全性和一致性。

class BankAccount:

def \_\_init\_\_(self, account\_number, balance):

self.\_\_account\_number = account\_number # 私有屬性

self.\_\_balance = balance # 私有屬性

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

def withdraw(self, amount):

if 0 < amount <= self.\_\_balance:

self.\_\_balance -= amount

def get\_balance(self):

return self.\_\_balance

# 私有方法，用於內部計算手續費

def \_\_calculate\_fee(self):

return max(self.\_\_balance \* 0.01, 5)

# 公開方法，用於提取現金並扣除手續費

def withdraw\_with\_fee(self, amount):

if 0 < amount <= self.\_\_balance:

fee = self.\_\_calculate\_fee()

self.\_\_balance -= (amount + fee)

return amount

# 創建銀行帳戶

account = BankAccount("123456", 1000)

# 存款和提款

account.deposit(500)

account.withdraw(200)

# 使用公開方法提取現金並扣除手續費

withdrawn\_amount = account.withdraw\_with\_fee(300)

print("提取金額（扣除手續費）：", withdrawn\_amount)

print("帳戶餘額：", account.get\_balance())

在這個範例中，BankAccount 類別具有兩個私有屬性 \_\_account\_number 和 \_\_balance，這些屬性被雙下劃線（\_\_）開頭，表示它們是**私有**的。這樣外部程式碼就不能直接訪問這些屬性。相反，我們提供了公開的方法 deposit、withdraw、get\_balance 和 withdraw\_with\_fee 來操作這些屬性，這些方法可以訪問私有屬性並進行適當的操作。

另外，我們還在類別中定義了一個私有方法 \_\_calculate\_fee，用於內部計算手續費。這個方法只能在類別內部被訪問，無法從外部呼叫。

這個範例演示了如何使用封裝的原則，保護數據的完整性，同時只提供必要的接口進行操作。這種設計方式可以確保銀行帳戶的一致性和安全性。

**抽象化（Abstraction）**

抽象化是將複雜的現實世界映射到簡單的模型，並隱藏不必要的細節。在程式設計中，抽象化通常是指將共同的特性和行為提取到基礎類別，然後在衍生類別中進行擴展。

以下是一個抽象化的範例，使用 Python 的類別來模擬不同類型的動物：

from abc import ABC, abstractmethod

class Animal(ABC):

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

self.name = name

@abstractmethod

def make\_sound(self):

pass

class Dog(Animal):

def make\_sound(self):

return "汪汪！"

class Cat(Animal):

def make\_sound(self):

return "喵喵！"

# 創建動物的物件

dog = Dog("旺財")

cat = Cat("小花")

# 使用物件的方法

print(dog.name + "發出聲音：" + dog.make\_sound())

print(cat.name + "發出聲音：" + cat.make\_sound())

在這個範例中，我們定義了一個抽象基礎類別 Animal，它繼承自 ABC 類別（Abstract Base Class），並定義了一個抽象方法 make\_sound。這個抽象方法在基礎類別中不實作，而是留給衍生類別來實現。衍生類別 Dog 和 Cat 都繼承自 Animal，並分別實作了 make\_sound 方法。

由於 Animal 是抽象類別，你無法創建它的實例。而衍生類別必須實作基礎類別的抽象方法，這樣才能被創建實例並使用。

這個範例展示了如何使用抽象化的概念，將共同的特性抽象到基礎類別中，然後在不同的衍生類別中實作具體的行為。這樣的設計方式讓程式碼更容易理解和擴展，同時也能保持一致性。

**多重繼承（Multiple Inheritance）**

多重繼承是指一個類別可以同時繼承自多個基礎類別，從而獲取多個基礎類別的特性和行為。在 Python 中，你可以通過列出多個基礎類別來實現多重繼承。

以下是一個多重繼承的範例：

class Swimmer:

def swim(self):

return "在水中游泳"

class Flyer:

def fly(self):

return "在空中飛行"

class Walker:

def walk(self):

return "在陸地上移動"

# 衍生類別 Penguin，同時繼承自 Swimmer 和 Walker

class Penguin(Swimmer, Walker):

def speak(self):

return "咕咕！"

# 衍生類別 Sparrow，同時繼承自 Flyer 和 Walker

class Sparrow(Flyer, Walker):

def speak(self):

return "嘰嘰！"

# 創建動物的物件

penguin = Penguin()

sparrow = Sparrow()

# 使用方法

print("企鵝：" + penguin.speak()) # 企鵝：咕咕！

print("企鵝：" + penguin.swim()) # 企鵝：在水中游泳

print("企鵝：" + penguin.walk()) # 企鵝：在陸地上移動

print("麻雀：" + sparrow.speak()) # 麻雀：嘰嘰！

print("麻雀：" + sparrow.fly()) # 麻雀：在空中飛行

print("麻雀：" + sparrow.walk()) # 麻雀：在陸地上移動

在這個範例中，我們定義了三個基礎類別 Swimmer、Flyer 和 Walker，分別代表可以在水中游泳、在空中飛行和在陸地上移動的動物特性。然後，我們創建了兩個衍生類別 Penguin 和 Sparrow，它們分別同時繼承了不同的基礎類別，實現了不同的特性。

Penguin 類別同時繼承自 Swimmer 和 Walker，因此可以在水中游泳和在陸地上移動。Sparrow 類別同時繼承自 Flyer 和 Walker，因此可以在空中飛行和在陸地上移動。

**多型（Polymorphism）**

多型（Polymorphism）是面向物件程式設計中的一個重要概念，它允許不同的類別使用相同的介面來執行不同的操作。多型讓你可以以一種統一的方式處理不同類型的物件，從而提高程式碼的靈活性和可讀性。

在 Python 中，多型通常是通過繼承和方法覆寫來實現。當不同的子類別繼承自同一基礎類別並覆寫相同名稱的方法時，這些子類別的物件可以以相同的方式呼叫這個方法，但實際上執行的是各自子類別中的實作。這就是多型的應用。

class Animal:

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

self.name = name

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "汪汪！"

class Cat(Animal):

def speak(self):

return "喵喵！"

class Parrot(Animal):

def speak(self):

return "嘰嘰！"

# 創建不同類型的動物

dog = Dog("旺財")

cat = Cat("小花")

parrot = Parrot("彩虹")

# 使用多型呼叫 speak 方法

animals = [dog, cat, parrot]

for animal in animals:

print(animal.name + "發出聲音：" + animal.speak())

在這個範例中，我們創建了不同類型的動物物件，並將它們放入一個列表 animals 中。然後，我們使用迴圈遍歷這個列表，每個動物物件都可以呼叫 speak 方法，但實際上執行的是該物件所屬類別的實作。這就是多型的效果，不同類型的物件可以使用相同的介面來進行操作。

多型讓程式碼更加靈活，並能夠更好地處理不同類型的物件，同時提高程式碼的可讀性和維護性。

**總結**

類別（Class）是面向物件程式設計（OOP）的核心概念之一，它允許你創建複雜的資料結構，包含屬性和方法，用於模擬現實世界中的實體和行為。以下是關於類別用法的總結：

定義類別： 使用 class 關鍵字定義類別，你可以在類別中定義屬性和方法。

屬性（Properties）： 類別的屬性是與該類別相關的資料，它們儲存物件的狀態。你可以使用 self 關鍵字來定義屬性，並在 **init** 方法中初始化它們。

方法（Methods）： 類別的方法是與該類別相關的函式，用於操作類別的屬性，執行特定的任務。

初始化方法（init）： 這是一個特殊的方法，用於初始化物件的屬性。它在創建物件時自動呼叫。

繼承（Inheritance）： 繼承允許你創建一個新的類別，基於現有的類別，並繼承其屬性和方法。這促進了程式碼重用和組織。

封裝（Encapsulation）： 封裝是隱藏物件的內部細節，只公開必要的介面。這可以防止外部程式碼直接訪問物件的屬性，確保資料的安全性。

抽象化（Abstraction）： 抽象化是提取類別的共同特性和行為，以形成一個簡化的模型。這提高了程式碼的可讀性和可維護性。

多重繼承（Multiple Inheritance）： 多重繼承允許一個類別同時繼承自多個基礎類別，獲取它們的特性和方法。

多型（Polymorphism）： 多型允許不同的子類別使用相同的介面來執行不同的操作，提高程式碼的靈活性。

使用範例： 通過定義不同類型的類別、創建物件、訪問屬性和呼叫方法，你可以在程式中模擬現實世界的情況。

總之，類別是一種強大的程式設計工具，它允許你組織程式碼、模擬現實世界中的概念，並提供了結構化和模組化的方式來開發複雜的應用程式。

**Python Abstract Base Classes 抽象類別**

a. 首先我們先導入一個package: import abc，很特別的名字

b. 再來，在我們想要抽象化的類裡面的參數，寫入abc.ABC 或 metaclass = abc.ABCMeta，就將它抽象化了

c. 抽象類裡面的function必須前面加上@abc.abstractmethod

d. 欲繼承的子類別，要在類參數裡寫入父類別的名稱

e. 記得: 繼承這個抽象類的子類別，必須有抽象類裡面的function，不然會出現錯誤，這個時候你就要再將這些function補上