前言

當我開始學習 Python 的時候，我很訝異它竟然這麼容易學習，一開始就可以寫一些基本的程式了，我試過幾種開發環境，每個環境我都可以快速地執行簡單的程式。

Python 的語法（syntax）非常簡單，並且沒有括號或分號要記住。除了記住使用 Tab 鍵（用來完成四個空格的縮排）之外，用 Python 寫程式很容易。

但是直到我玩了幾個星期 Python 之後，我才開始看到這門語言到底有多複雜，以及您可以用它做多少事情。Python 是一種物件導向程式語言，可以很容易地建立包含自己資料的類別，不需要大量繁雜的語法。

事實上，我開始嘗試編寫一些幾年前用 Java 編寫的程式，我很訝異它們在 Python 中竟然如此簡潔。藉由強大的 IDE，避免了很多常見的錯誤。

當我意識到我可以用 Python 快速的完成很多事情時，我也覺得是時候寫一本關於可以用 Python 編寫強大程式的書了，這促使我把自己在幾年前最初編寫的 23 種經典設計模式，重新編寫為全新又簡潔好讀的版本。

成果就是這本書，它說明了物件導向程式的基礎知識、視覺化程式開發，以及如何使用所有的經典模式。您可以在 GitHub 上找到完整的程式碼：

https://github.com/jwcnmr/jameswcooper/tree/main/Pythonpatterns

本書旨在幫助 Python 軟體開發人員拓展物件導向程式設計 (OOP) 和相關設計模式的知識。

- 如果您是 Python 新手，但有其他語言的經驗，您可以先複習第 31 章到第 35 章，然後從第 1 章開始閱讀。
- 如果您有 Python 經驗，但想了解物件導向程式設計和設計模式，請從第 1 章開始。如果您願意，可以跳過第 2 章和第 3 章，直接閱讀本書的其餘部分。
- 如果您是一般程式新手，請花一些時間閱讀第 31 章到第 35 章，嘗試其中的一些程式，然後從第 1 章開始學習物件導向程式設計和設計模式。

碁峯 www.gotop.com.tw

您可能會發現 Python 是您學過最簡單的語言，也是您在設計模式中編寫物件最輕鬆的語言。您將了解它們的用途以及如何在自己的工作中使用它們。

無論如何，這些頁面中介紹的物件導向程式設計方法，可以幫助您編寫更好、可重複使用的程式碼。

碁峰
www.gotop.com.tw

本書結構

本書分為五個部分。

第一部分　入門

設計模式本質上是在描述物件如何有效地互動。本書首先在第 1 章「物件入門」中介紹物件，並提供圖像化範例來清楚地說明模式的工作原理。

第 2 章「Python 中的視覺化程式開發」和第 3 章「資料表視覺化程式設計」介紹了 Python tkinter 函式庫，它為您提供了一種以最小複雜度來建立窗口、按鈕、清單、表格等的方法。

第 4 章「什麼是設計模式？」透過探索設計模式是什麼，來開始討論設計模式。

第二部分　建立型模式

第二部分首先概述了「四人幫」命名為建立型模式的第一組模式。

第 5 章「工廠模式」描述了基本的工廠模式，它是後面三種工廠模式的簡單基礎。在本章中，您將建立一個 Factory 類別，該類別根據資料本身，來決定使用哪一個相關類別。

第 6 章「工廠方法模式」探討了工廠方法。在這種模式中，沒有一個類別決定實例化哪個子類別。事實上，父類別將實例化的決定推遲到每個子類別。

第 7 章「抽象工廠模式」討論了抽象工廠模式的內容。當想要傳回幾個相關的物件類別中的其中一個，可以使用此模式，每個類別都可以根據請求傳回幾個不同的物件。換句話說，抽象工廠是一個工廠物件，它傳回多組類別的其中一個。

碁峰
www.gotop.com.tw

第 8 章「單例模式」主要著眼於單例模式，它描述了一個類別，其中不能有多個實例。它提供對該實例的單一全域存取點。您並不會經常使用這種模式，但知道如何編寫它會很有幫助。

在第 9 章「建造者模式」中，您會看到建造者模式將複雜物件的構造與其視覺化表示分開，因此可以根據程式的需要建立幾種不同的表示。

第 10 章「原型模式」說明建立類別的實例既耗時又複雜時，如何使用原型模式。您無須建立更多實例，只需複製原始實例，並根據需要進行修改。

第 11 章「建立型模式總結」對第二部分的模式進行了總結。

第三部分　結構型模式

第三部分從對結構模式的簡短討論開始。

第 12 章「適配器模式」介紹適配器模式，該模式用於將一個類別的程式開發接口，轉換為另一個類別的程式開發接口。當您希望不相關的類別在單一程式中一起工作時，適配器很有用。

第 13 章「橋接模式」採用了類似橋接的模式，旨在將類別的介面與實作分開。這能夠在不更改客戶端程式碼的情況下更改或替換實作。

第 14 章「組合模式」深入研究了元件可能是單一物件或表示物件集合的系統，組合模式旨在適應前述這兩種情況，通常採用樹狀結構。

在第 15 章「裝飾者模式」中，我們將了解裝飾者模式，它提供了一種無須建立新衍生類別，即可修改單一物件行為的方法。儘管這可以應用於按鈕等視覺物件，但在 Python 中最常見的用途，是建立一種修改單一類別實例行為的巨集。

在第 16 章「門面模式」中，我們學習使用門面模式來編寫一個簡化的介面，以處理可能過於複雜的程式碼。本章將討論一個和多個不同資料庫的介面。

第 17 章「享元模式」描述享元模式的內容，它使您能夠藉由將一些資料移到類別外來減少物件的數量。當您有同一個類別的多個實例時，您可以考慮這種方法。

碁峰 www.gotop.com.tw

第 18 章「代理模式」介紹了代理模式，當您需要用一個更簡單的物件，來表示一個複雜或耗時的物件時，就會使用該模式。如果建立物件需要耗費大量時間或電腦資源，代理模式可以讓您推遲，直到您需要實際物件。

第 19 章「結構型模式總結」，總結了這些結構型模式的內容。

第四部分　行為型模式

第四部分概述行為型模式的內容。

第 20 章「責任鏈模式」，著重於責任鏈模式如何透過將請求從一個物件傳遞到鏈中的下一個物件直到請求被識別，從而實現物件之間的解耦。

第 21 章「命令模式」展示了命令模式如何使用簡單物件來表示軟體命令的執行。此外，此模式使您能夠支援日誌記錄和可取消的操作。

第 22 章「解譯器模式」著眼於解譯器模式，它提供了如何建立一個小的執行語言並包含在程式中的定義。

在第 23 章「疊代器模式」中，我們探討了著名的疊代器模式，它描述了在資料項集合中移動的正式方法。

第 24 章「中介者模式」介紹了重要的中介者模式。此模式定義了如何透過使用單獨的物件，來簡化物件之間的通信，讓所有物件不必相互了解。

第 25 章「備忘錄模式」，保存物件的內部狀態，以便以後恢復它。

在第 26 章「觀察者模式」中，我們將了解觀察者模式，它使您能夠定義在程式狀態發生變化時通知多個物件的方式。

第 27 章「狀態模式」描述了狀態模式，它允許物件在其內部狀態發生變化時修改其行為。

第 28 章「策略模式」描述了策略模式，它與狀態模式一樣，無須任何單一的條件語句，即可在演算法之間輕鬆切換。狀態模式和策略模式之間的區別在於使用者通常在多種策略中，選擇一種來應用。

碁峰 www.gotop.com.tw

在第 29 章「模板模式」中，我們將了解模板模式。這種模式形式化了在類別中定義演算法的想法，但留下一些細節在子類別中實作。換句話說，如果您的基礎類別是一個抽象類別，就像這些設計模式中經常發生的那樣，您使用的是模板模式的簡單形式。

第 30 章「拜訪者模式」探討了拜訪者模式，該模式在物件導向模型上扭轉局面，並建立一個外部類別，來處理其他類別中的資料。如果有少量類別的大量實例，並且您想要執行一些涉及所有或大部分類別的操作，這會很有用。

第五部分　Python 簡介

在本書的最後一部分，我們提供了 Python 語言的簡單摘要。如果您只是暫時需要熟悉 Python，這將讓您可以快速上手。它也能夠徹底指導初學者。

在第 31 章「Python 中的變數及語法」中，我們回顧了基本的 Python 變數和語法；在第 32 章「Python 中的條件判斷」中，說明了程式可以做出決策的方式。

在第 33 章「開發環境」中，我們簡要總結了最常見的開發環境。

在第 34 章「Python 中的集合和檔案」中，我們將討論陣列和檔案。

最後在第 35 章「函式」中，我們將討論如何在 Python 中使用函式。

享受編寫設計模式和學習強大的 Python 語言之來龍去脈！

碁峰 www.gotop.com.tw

第 1 章

物件入門

類別（Class）在 Python 中占有重要地位，也是物件導向（object-oriented programming）的重要元素，有些書籍將類別放在較後面的章節，但因為幾乎整個 Python 中的元件（component）都是物件，本書會在一開始就介紹它們，不要跳過這個部分，因為後面每一章都會用到。

幾乎所有 Python 中的元件（component）都是物件。

- 物件包含資料（data），而且有方法可以存取並改變資料。

舉例來說，字串（strings）、串列（lists）、元組（tuples）、集合（sets）和字典（dictionaries）都是物件，複數也是物件。他們都有一個函式（function）叫做*方法*（*method*），可以讓您存取或者是改變資料的值。

```
list1 = [5, 20, 15, 6, 123] # 建立一個 list
x = list1.pop()             # 移除最後加入的值 , x =123
```

這就是使用常見 Python 物件的方式，但是要怎麼有自己的物件呢？

- 類別使您可以建立新物件。

一個類別（Class）可能看起來有點像函式（function），它們最大的不同在於類別可以有多個*實例*（*instances*），每個實例包含不同的資料，類別可以包含許多函式，每個函式存取該實例類別上的資料，每個類別的實例通常被稱為*物件*（*object*），每個函式通常被稱為*方法*（*method*）。

很多時候，我們用類別來表示真實世界的概念，像是商店、客戶和銀行。請使用一個可以描述物件的類別名稱，而不是類似可愛的 `Dog` 類別，讓我們建立一個包含有用名稱的類別來描述員工，我們的 `Employee` 類別，包含員工的姓名、薪水、福利以及員工編號。

碁峰 www.gotop.com.tw

```
class Employee():
    def __init__(self, frname, lname, salary):
        self.idnum: int          # 占位
        self.frname = frname     # 儲存名稱
        self.lname = lname
        self._salary = salary    # 儲存薪水
        self.benefits = 1000     # 儲存福利

    def getSalary(self):         # 取得薪水
        return self._salary
```

每個員工的值都是在 _init_ 方法中設定的，當我們建立 Employee 類別時，該方法會自動調用，self 前綴代表想存取該實例類別中的變數，同一個類別中不同實例的同一變數可以有不同的值，在同一個類別中，可以使用 self 前綴來存取所有的變數和方法。

類別 __init__ 方法

當建立一個類別的實例時，只需建立一個變數，並傳入參數：

```
fred = Employee('Fred', 'Smythe', 1200)
sam  = Employee('Sam', 'Snerd', 1300)
```

變數 fred 和 sam 是 Employee 類別中的*實例*，有指定的值給 name 和 salary，我們可以建立同樣的 Employee 實例類別，每個都對應到一個員工。

- 一個類別可以有很多實例，每個實例都有不同的值。
- 每個實例也可以稱為一個*物件*。
- 類別中的函式稱為*方法*。

類別中的變數

Employee 類別包含變數 first name、last name、salary、benefits 以及 ID number，我們用 getSalary 方法，但是為什麼不直接存取它呢？在很多相似的語言中，類別中的變數是私有（private）或隱藏的，所以您需要一個存取方法（accessor method）

碁峯 www.gotop.com.tw

去取出這些值，不過 Python 讓您可以做任何想做的事，您不需要使用 getSalary 或是一個屬性（property），可以撰寫：

```
print(fred._salary)
```

來直接取得它的值，那為什麼還需要使用存取子函式（accessor function）呢？部分是為了強調類別中的變數是私有的，實例可能會改變，但存取子函式將保持不變。而且，在某些情況下，存取子函式傳回的值可能必須在那個時候計算出來。

有一個 Python 的約定，使用雙底線開頭命名這些私有變數，約定主要強調您不打算直接存取它，這讓他們更難不小心輸入值。很多開發環境，像是 PyCharm，在您輸入

```
fred.
```

的時候甚至都不會提示這些變數的存在，查看可能出現的變數和方法時，帶有雙底線開頭的變數和方法不會顯示，如果您堅持的話，還是可以存取它們，這只是一個約定，不是一個強制的語法要求。

類別的集合

現在讓我們想想如何將這些 Employee 類別存起來，您可能會放在資料庫中，不過在程式中，使用某種集合似乎是個好主意。我們將定義一個 Employee 類別，將員工（employees）保存在字典（dictionary）中，每個人都有自己的 ID，它看起來像這樣：

```
# 包含員工字典，以 ID 為鍵值
class Employees:
    def __init__(self):
        self.empDict = {}     # 員工字典
        self.index = 101      # 起始編號

    def addEmployee(self, emp):
        emp.idnum = self.index  # 設定 ID
        self.index += 1         # 移至下個 ID
        self.empDict.update({emp.idnum: emp}) # 加入
```

在上面的 Employee 類別中，我們建立了一個空字典和一個起始 ID 編號，每次我們將一個員工加到該類別中時，它會增加索引值。

碁峰 www.gotop.com.tw

我們在名為 HR 的外部類別中建立類別：

```
# 建立 Employees 集合
class HR():
    def __init__(self):
        self.empdata = Employees()
        self.empdata.addEmployee(
                    Employee('Sarah', 'Smythe', 2000))
        self.empdata.addEmployee(
                    Employee('Billy', 'Bob', 1000))
        self.empdata.addEmployee(
                    Employee('Edward', 'Elgar', 2200))

    def listEmployees(self):
        dict = self.empdata.empDict
        for key in dict:
            empl= dict[key] # 取得實體
            # 印出它們
            print (empl.frname, empl.lname,
                                 empl.salary)
```

- 您可以將類別實例，保留在其他類別中。

繼承（Inheritance）

繼承是物件導向程式設計中另一個強大的工具，不僅可以在同一個類別中，建立不同實例，還可以建立**衍生類別**（*derived classes*），這些新類別具有父類別的所有屬性以及添加的任何其他屬性，但是請注意，衍生類別的 __init__ 方法必須調用父類別的 __init__ 方法。

在一個真正的公司中，我們可能還有其他類型的員工，包括有領取報酬（相同或更少）、但沒有獲得福利的員工。與其建立一個全新的類別，我們不如從 Employee 類別**衍生**一個新的 TempEmployee 類別。新類別的方法都一樣，所以不必重新編寫程式碼，只需要寫那些新的部分。

```
# 臨時工沒有福利
class TempEmployee(Employee):
    def __init__(self, frname,lname, idnum):
        super().__init__(frname, lname, idnum)
        self.benefits = 0
```

碁峰 www.gotop.com.tw

使用修改後的方法建立的衍生類別（Derived Classes）

您可以在物件導向程式中使用的一個技巧是建立衍生類別，其中一個或多個方法做的事情略有不同，這稱為多型（*polymorphism*），這是一個價值 50 美元的單字，用於改變新類別的形狀。

例如，我們可以從名為 Intern 的類別中建立另一個類別。實習生沒有福利且工資低。因此，我們編寫了一個新的衍生類別，它的 setSalary 方法檢查工資以確保它不超過上限。（當然，我們真的不認為這是一個好主意。）

```
# 實習生沒有福利，並且薪水比較少
class Intern(TempEmployee):
    def __init__(self, frname, lname, sal):
        super().__init__(frname, lname, sal)
        self.setSalary(sal) # 薪水上限

    # 限制實習生薪水
    def setSalary(self, val):
        if val > 500:
            self._salary = 500
        else:
            self._salary = val
```

多重繼承（Multiple Inheritance）

與 Java 和 C#（但與 C++ 相似）不同，Python 能夠建立從多個基礎類別繼承的類別。這可能看起來很令人困惑，但大多數人建立了一個類別層次結構，其中一些類別可能具有與其他類型的類別相同的一兩個方法。我們稍後會在第 21 章「命令模式」中看到，我們常常以這種方式使用命令（Command）類別。

碁峰 www.gotop.com.tw

假設我們的一些員工是優秀的公開演講者。我們可以建立一個單獨的類別，表明他們可以被邀請進行演講，並且可以因此獲得獎勵。

```
# 代表公開演講者的類別
class Speaker():
    def inviteTalk(self):
        pass
    def giveTalk(self):
        pass
```

這個例子暫時省略了實作細節，但是我們可以建立一個新的 Employee 衍生類別，該類別也衍生自 Speaker：

```
class PublicEmployee(Employee, Speaker):
    def __init__(self, frname, lname, salary):
        super().__init__(frname, lname, salary)
```

現在我們可以在每個類別中建立一組員工：

```
class HR():
    def __init__(self):
        self.empdata = Employees()
        self.empdata.addEmployee(
                   Employee('Sarah', 'Smythe',2000))
        self.empdata.addEmployee(
                   PublicEmployee('Fran', 'Alien',3000))
        self.empdata.addEmployee(
                   TempEmployee('Billy', 'Bob', 1000))
        self.empdata.addEmployee(
                   Intern('Arnold', 'Stang', 800))

    def listEmployees(self):
        dict = self.empdata.empDict
        for key in dict:
            empl= dict[key]
            print (empl.frname, empl.lname,
                          empl.getSalary())
```

請注意，雖然其中三個是衍生類別，但它們仍然是 Employee 物件，您可以像上面一樣列出它們。

- 衍生類別允許您建立具有不同屬性或計算（computation）的相關類別。

碁峯 www.gotop.com.tw

畫一個矩形和一個正方形

我們來看看最後一個繼承（inheritance）範例，這個例子是用 Canvas 這個物件的函式（function）來畫出一個矩形及一個正方形，Canvas 是 tkinter 函式庫的視覺物件，我們會在之後的章節繼續使用它，現在我們用 Canvas 的 create_rectangle 方法來畫一個矩形。

create_rectangle 方法有四個參數（x1,y1,x2,y2），不過我們現在要建立一個使用（x,y,w,h）的方法，讓轉換在名為 Rectangle 的類別中進行。

```
# 在畫布 (canvas) 上繪製矩形
class Rectangle():
    def __init__(self, canvas):
        self.canvas = canvas  # 複製 canvas 的參考

    def draw(self, x, y, w, h):  # 畫矩形
        # 使用 x1,y1, x2,y2 畫矩形
        self.canvas.create_rectangle(x, y, x+w, y+h)
```

結果如圖 1.1。

圖 1-1　在畫布上畫一個矩形

碁峰 www.gotop.com.tw

假設我們接著要畫一個正方形，我們可以有一個類別（Square）來繼承矩形（Rectangle）類別，快速地畫出正方形：

```
# Square 繼承自 Rectangle
class Square(Rectangle):
    def __init__(self, canvas):
        super().__init__(canvas)

    def draw(self, x, y, w):
        super().draw( x, y, w, w) # 畫一個正方形
```

現在我們將正方形的邊長輸入 Rectangle 兩次，一次是寬度，一次是高度。

```
def main():
    root = Tk()                 # 圖形函式庫
    canvas = Canvas(root)       # 建立一個 Canvas 實例
    rect1 = Rectangle(canvas)   # 加入一個矩形
    rect1.draw(30, 10, 120, 80) # 繪製矩形

    square = Square(canvas)     # 建立一個正方形
    square.draw(200, 50, 60)    # 繪製正方形
```

結果如圖 1.2 所示。

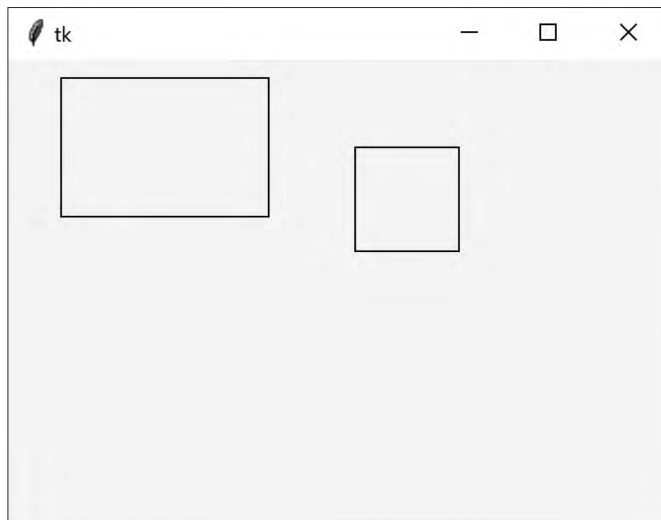

圖 1-2　畫面上一個矩形和一個正方形，皆繼承自 rectangle

碁峰 www.gotop.com.tw

變數的可見性

Python 程式中變數的可見性有四個級別：

- 全域變數（不明智）
- 類別裡面的變數
- 類別程式碼中的實例變數
- 區域變數僅在函式內（其他地方不可見）

想想看這個簡單程式的開頭：

```
""" 變數存取範例 """
badidea = 2.77 # 全域變數

class ShowData():
    localidea = 3.56      # 類別變數

    def __init__(self):
        self._instvar = 5.55  # 實例變數
```

全域變數 badidea 可以被任何類別中的任何函式存取，更糟糕的是，它可以被程式的任何部分修改。人們有時將全域變數用於常數，但使用類別變數更容易控制且不易出錯。

在前面的範例中，localidea 是類別最上方的變數，但不是類別中任何方法的一部分。該類別的成員和其他類別的成員，可以透過類別名稱和變數名稱來存取它：

```
print(ShowData.localidea)
```

他們也可以更改它，但這可能不是一個好習慣。

實例變數對於類別的每個實例都是唯一的，並且是藉由在變數名稱前加上 self 前綴來建立的。

```
def __init__(self):
    self._instvar = 5.55  # 實例變數
```

透過建立一個變數 _instvar，我們表達了不應在類別外部存取該變數。如果嘗試用以下方式存取它，各種開發環境都會警告您。

```
print (ShowData._instvar)
```

碁峰 www.gotop.com.tw

取得這些實例變數的常用方法是使用 getter 和 setter 方法：

```
# 回傳實例變數
def getInstvar(self):
     return self._instvar

# 設定值
def setInstvar(self, x):
    self._instvar = x
```

屬性（Property）

您還可以使用屬性裝飾器（property decorator）來取得和儲存實例變數：

```
# getters 和 setters 可以保護
# 實例變數的使用
    @property
    def instvar(self):
        return self._instvar


    @instvar.setter
    def instvar(self, val):
        self._instvar = val
```

這些裝飾器能夠用方法來存取或更改實例變數，在值可能超出範圍時保護實際值。

```
print(sd.instvar)        # 使用 getter
sd.instvar = 123         # 使用 setter 去更改
```

區域變數

類別中函式內的變數僅存在於該函式內。例如以下範例 x 和 i 都是本地的，只在該函式內，不能在它之外存取：

```
def addnums(self):
    x = 0                    # i 和 x 是本地的
    for i in range(0, 5):
        x += i
    return x
```

碁峰 www.gotop.com.tw

Python 中的型別

Python 中的變數是在執行階段動態輸入的，而不是事先聲明型別。Python 從分配給變數的值推斷型別，當型別有衝突時，有時會導致執行階段問題。這種方法稱為鴨子型別，基於古老的格言，「如果它看起來像鴨子，叫起來像鴨子，那就是鴨子。」

在 3.8 版中，Python 添加了 *type hints* 來告訴靜態型別檢查預期的型態。靜態型別檢查不是 Python 本身的一部分，但大多數開發環境（例如 PyCharm）會自動執行它，並提示可能的錯誤。

您可以宣告並傳回每個參數的型別，如下所示：

```
class Summer():
    def addNums(self, x: float, y: float) ->float:
        return x + y
```

更令人驚豔的是，可以擁有兩個或多個名稱相同但參數不同的函式：

```
    def addNums(self, f: float, s: str)->float:
        fsum = f + float(s)
        return fsum
```

Python 將根據參數調用正確的函式，無論是放入兩個浮點數，還是一個浮點數和一個字串：

```
sumr = Summer()
print(sumr.addNums(12.0, 2.3))
print(sumr.addNums(22.3, "13.5"))
```

然後印出來：

```
14.3
35.8
```

這稱為**多型**（*polymorphism*），原意為採取不同形式的能力。在這裡代表可以擁有多個名稱相同但參數不同的方法，您可以根據選擇的參數調用需要的方法。這個特性在 Python 中普遍使用。

但是，如果調用 addNums(str, str)，會發現 PyCharm 和其他型別檢查器將此標記為錯誤，因為沒有這樣的方法，會收到以下錯誤訊息：

```
Unexpected types (str, str)
```

碁峯 www.gotop.com.tw

總結

本章涵蓋了物件導向程式設計的所有基礎知識，所以這裡做個總結：

- 可以使用 `class` 關鍵字跟大寫的類別名稱來建立類別。
- 類別包含資料，一個類別的每個實例可以保存不同的資料。這就是所謂的*封裝*。
- 可以建立從其他類別衍生的類別。在類別名稱後面的括號中，指示新衍生類別的類別名稱，這稱為*繼承*。
- 可以建立一個衍生類別，其方法在某種程度上不同於基礎類別。這稱為*多型*。
- 也可以建立包含其他類別的類別。我們將在下面看到一個範例章節。

Github 範例程式碼

記得您可以在 GitHub 上的 jameswcooper/pythonpatterns 找到所有範例。

- BasicHR.py：包含沒有衍生類別的 `Employees`
- HRclasses.py：包含兩個衍生類別
- Speaker.py：包含 `Speaker` 類別
- Rectangle.py：繪製正方形和矩形
- Addnumstype.py：多型函式調用

如果您不熟悉 GitHub，它是一個免費的軟體儲存庫，用於共享任何人都可以使用的程式碼。要開始使用，請打開瀏覽器，並輸入 GitHub.com，再點擊 Sign Up 註冊。您將需要建立使用者帳號和密碼，並提交電子郵件地址以進行驗證。之後您可以搜尋任何程式碼倉庫（比如 jameswcooper），並下載想要的程式碼，該網站上也有完整的手冊。

登錄後，您也可以直接在這找到範例：
https://github.com/jwcnmr/jameswcooper/tree/main/Pythonpatterns。

碁峯 www.gotop.com.tw