簡介

這本書的主題是關於**物件導向程式設計（OOP，object-oriented programming）**的軟體開發技術以及如何讓它與 Python 搭配運用。在 OOP 之前，程式設計師使用的開發方法是稱為**程序式程式設計（procedural programming）**，也稱為**結構化程式設計（structured programming）**，此方法涉及建構一組函式（程序）並透過呼叫這些函式來傳遞資料。而本書主講的 OOP 範式為程式設計師提供了一種更有效率的做法，可以把程式碼和資料組合成具有高度可重用的內聚單元。

在準備編寫本書時，我對現有的文獻和網路上的視訊教學進行了廣泛的研究，特別關注了解釋說明這個重要且範圍廣泛的方法論。我發現老師和作者們通常會從定義某些關鍵術語開始：**類別（class）**、**實例變數（instance variable）**、**方法（method）**、**封裝（encapsulation）**、**繼承（inheritance）**、**多型（polymorphism）**等等。

雖然這些都是重要的觀念，而我也會在本書中深入介紹這些觀念，但我會以不同的方式著手：會利用「我們要解決什麼問題？」這個題目來思考，也就是說，如果 OOP 是解決方案，那麼問題是什麼呢？為了回答這個題目，我會先展示一些以程序式程式設計所建構的程式範例，並確定這種做法的複雜性，然後再向您展示物件導向的做法，讓您了解怎麼樣才能讓此類程式的建構變得更容易。此外這種程式本身很容易維護。

本書適用對象

本書適用於已經熟悉 Python 並能使用 Python 標準程式庫中基本函式的讀者。我假設您已了解 Python 語言的基本語法，並且能夠使用變數、指定值陳述句、if/elif/else 陳述句、while 和 for 迴圈、函式和函式呼叫、串列、字典等等語法。如果您對所有這些基本概念還不太熟悉，那麼我建議您閱讀我之前寫的書《Learn to Program with Python 3》（Apress 出版）。

本書的內容算是中階的程度，因此有許多更高階的主題是不會討論的，例如，為了保持本書的實用性，我不會詳細介紹 Python 的內部實作。為了讓書中的知識簡單明瞭，並專注於掌握 OOP 的技術，書中的程式範例都是使用 Python 的常用功能來編寫的，雖然 Python 還有更高階、更簡潔的寫法，但超出了本書的適用範圍。

我所介紹的 OOP 底層細節是獨立於程式語言的，但會指出 Python 和其他 OOP 語言之間的差異。透過本書所學習到的 OOP 風格的程式設計基礎知識和技術，也能夠輕鬆地套用到其他 OOP 語言中。

Python 版本與安裝

本書中的所有範例程式碼都是使用 Python 3.6 到 3.9 版本編寫和測試的。所有範例程式都適用於 3.6 或更高版本。

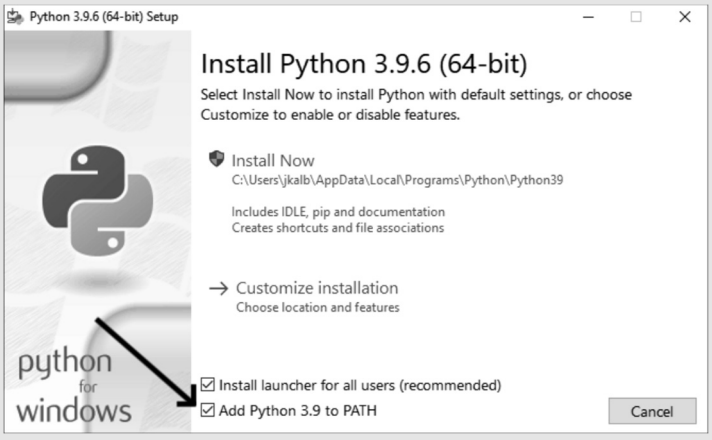
Python 可在 https://www.python.org 網站免費取得。如果您還沒有安裝 Python，或者您想升級到最新版本，請連到該網站，找到「Download」標籤，然後按下 **Download** 鈕，這樣就能把安裝檔下載到您的電腦。連按二下剛才下載的檔案就能安裝 Python。

NOTE

我知道「PEP 8 – Style Guide for Python」指引中其對變數和函式名稱是使用 snake_case 大小寫的慣例來命名。但是，在 PEP 8 出來之前，我已經使用駝峰式（camelCase）命名慣例很多年，而且在我的職涯中已經習慣了這種方式。因此，本書的所有變數和函式名稱都是以駝峰式的慣例來命名。

Windows 版的安裝

如果您在 Windows 系統上安裝 Python，則需要記得勾選設定一個重要的選項。在執行安裝步驟的畫面中，您應該會看到如下對話方塊：



對話方塊的底部有一個「Add Python 3.9 to PATH」核取方塊，請務必勾選（預設是沒有勾選）。此設定會讓 pygame 套件（本書後面會介紹）正確安裝，並讓它能正確工作。

我會怎麼樣解釋說明 OOP 這個觀念？

前幾章的例子使用文字式互動的 Python 程式來解說，這些範例程式會從使用者那裡獲取輸入資料，並以純文字的形式向使用者輸出資訊。我會透過展示怎麼開發這種文字式實體物件模擬的程式碼，以此來講述和介紹 OOP 觀念。我們一開始是以電燈開關、調光開關和電視遙控器為例，並把它們製作成「物件」來表示，隨後講解怎麼使用 OOP 來模擬銀行帳戶和控制多個帳戶。

在學習了 OOP 的基礎知識後，我會介紹 pygame 模組，這個模組允許程式設計師開發 GUI（圖形使用者介面）的遊戲程式和應用程式。在 GUI 型式的程式中，使用者可以直觀地與按鈕、核取方塊、文字輸入和輸出欄位以及其他對使用者友善的小元件進行互動。

我選擇把 pygame 與 Python 結合運用，因為這種組合讓我能使用螢幕上的元素，以高度視覺化的方式來展示 OOP 的觀念。pygame 的可攜性很強，幾乎可以在所有平台和作業系統上執行。書中所有使用 pygame 套件的範例程式都是使用最近發布的 pygame 2.0 版本來編寫和測試的。

我開發了一個名為 pygwidgets 的套件，它與 pygame 一起搭配並實作了許多基本 widgets 小工具，所有這些小工具都是使用 OOP 方法建構的。我會在本書後面介紹這個套件，並提供可以執行和試用的範例程式碼。這種方法會讓讀者學習到物件導向關鍵概念的真實和實用範例，同時結合這些技術來製作有趣、好玩的遊戲程式。我還會介紹我開發的 pyghelpers 套件，其中的程式碼能用來協助您編寫開發更複雜的遊戲程式和應用程式。

本書中顯示的所有範例程式碼和相關資源都可以從 No Starch 的官方網站單獨下載：https://www.nostarch.com/object-orientated-python/。

這些程式碼也可以從我的 GitHub 倉庫中逐章取得：https://github.com/IrvKalb/Object-Oriented-Python-Code/。

本書內容

本書分為四篇。

Part 1 篇介紹物件導向程式設計：

- 第 1 章回顧了程序式程式設計的做法。這裡會展示如何實作文字型的紙牌遊戲程式，並模擬銀行系統對一個或多個帳戶執行相關操作。在此過程中，我會討論程序式方法的常見問題。
- 第 2 章介紹了類別和物件，並展示如何使用類別在 Python 中表示現實世界的物件，並以電燈開關或電視遙控器為例來示範。您將會學到怎麼用物件導向的方法來解決第一章中強調的問題。
- 第 3 章介紹了兩個心智模型，當您在 Python 中建立物件時，可以利用它們來思考幕後發生的事情。我們會使用 Python Tutor 逐步執行程式碼並觀察物件是怎麼建立的。

- 第 4 章介紹物件管理器物件的概念，示範處理同一類型多個物件的標準方法。我們會使用類別來擴充銀行帳戶程式的模擬範例，並展示如何使用例外來處理錯誤。

Part 2 篇把焦點放在使用 pygame 建構 GUI 程式：

- 第 5 章介紹了 pygame 套件和程式設計的事件驅動模型。我們會建構一些簡單的程式來幫助您開始在 Windows 視窗中放置影像圖型並處理鍵盤和滑鼠的輸入，隨後還會開發更為複雜的球彈跳 ball-bouncing 程式。
- 第 6 章更詳細介紹在 pygame 程式怎麼運用 OOP 的概念。我們將以 OOP 風格重寫 ball-bouncing 程式，並開發一些簡單的 GUI 元素。
- 第 7 章介紹 pygwidgets 模組，此模組中含有許多標準 GUI 元素（按鈕、核取方塊等）的完整實作，每一種都可當作一個類別來進行開發。

Part 3 篇深入探討 OOP 的主要原則：

- 第 8 章討論封裝，這個概念涉及對外部程式碼隱藏實作細節並將所有相關方法放在一個地方——那就是放在「類別」中。
- 第 9 章介紹多型——多個類別可以有相同名稱的方法——並展示了多型如何讓您能夠呼叫多個物件中的方法，而且不必知道每個物件的型別。我們會建構一個 Shapes 程式來示範這個概念。
- 第 10 章介紹繼承，此概念允許您建立一組子類別，而這些子類別都共用內建在基礎類別中的通用程式碼，不必重新發明輪子再重複建立類似的類別功能。我們會介紹繼承應用的實際範例，例如實作一個只接受數字的輸入欄位，然後重寫之前的 Shapes 範例程式來使用此功能。
- 第 11 章透過討論一些其他重要的 OOP 主題來總結這一篇的內容，這些主題大多與記憶體管理相關。我們會觀察一個物件的生命週期，並以一個實例來解說，這裡會建構一個小型的氣球遊戲程式來示範。

Part 4 篇探討了與在遊戲開發中使用 OOP 相關的幾個主題：

- 第 12 章展示如何把第 1 章中開發的紙牌遊戲重新建構成以 pygame 為基礎的 GUI 程式。這一章還會展示如何建構可在開發其他紙牌遊戲中能重複使用的 Deck 和 Card 類別。
- 第 13 章討論了時間的處理。我們會開發不同的計時器類別，允許程式在持續執行中同步檢查給定的時間限制。
- 第 14 章解說可用於顯示影像序列的動畫類別。我們會探究兩種動畫技術：從一組單獨的影像檔來建構動畫，以及從單個拼合圖檔中擷取和使用多個影像。
- 第 15 章解釋了狀態機的概念，狀態機可以用來表示和控制程式的流程。另外還解說場景管理器，您可以使用它來建構具有多個場景的程式。為了示範其中每種程式的用法，我們會建構兩個版本的 Rock、Paper、Scissors 遊戲程式。
- 第 16 章討論了不同類型的模態互動對話方塊，這是另一個重要的使用者互動功能。隨後會逐步建構一個名為 Dodger 的全功能電玩遊戲程式，此程式是以 OOP 為基礎來建構的，該遊戲程式示範了書中所講述的許多技術。
- 第 17 章介紹了設計模式的概念，重點介紹說明了 MVC 模式（Model View Controller pattern），隨後製作了一個擲骰子的程式，該程式是使用 MVC 模式允許使用者以多種不同的方式來視覺化處理和呈現資料。另外這裡也為本書做簡短的學習總結。

開發環境

在本書中，您只需要最低限度地使用命令列來安裝軟體。所有安裝說明都會清楚地寫出來，因此無需學習其他命令列的語法。

我強烈建議使用互動式開發環境（IDE，interactive development environment），而不是使用命令列進行開發。IDE 為我們處理了底層作業系統的許多細節，允許我們對單個程式來編寫、編輯和執行程式碼。IDE 通常是跨平台的，允許程式設計師輕鬆地從 Mac 移到 Windows 的電腦，反之亦然。

書中的簡短範例程式可以在安裝 Python 所內附的 IDLE 開發環境中執行。IDLE 使用起來非常簡單，很適合編寫單個檔案的程式。當我們遇到需要使用多個 Python 檔案的更複雜的程式時，我建議您使用更強大的開發環境來配合，我使用的是 JetBrains PyCharm 開發環境，它可以更輕鬆地處理多個檔案的程式專案。社群版可從網站：https://www.jetbrains.com/ 免費取得，我強烈推薦它。PyCharm 還有一個完全整合的 debugger，在編寫大型程式時非常有用。有關如何使用 debugger 的更多資訊，請參閱 YouTube 視訊「Debugging Python 3 with PyCharm」，網址為 https://www.youtube.com/watch?v=cxAOSQQwDJ4&t=43s/。

Widgets 小工具和範例遊戲程式

本書介紹並提供了兩個 Python 套件：pygwidgets 和 pyghelpers。使用這些套件，您就能夠建構完整的 GUI 程式。但更重要的是，您會了解各個 widgets 小工具是怎麼寫成一個類別並當作物件來使用。

書中的範例遊戲程式的開發會搭配結合各種 widgets 小工具，一開始會以相對簡單的例子來說明，隨後範例程式逐漸加強而變得複雜。第 16 章會引導您開發和實作一個功能齊全的電玩遊戲程式，還能儲存遊戲得分記錄到檔案中。

學習到本書的最後，您應該能夠寫出自己的遊戲程式，像紙牌遊戲程式、或是像 Pong、Hangman、Breakout、Space Invaders 等風格的電玩遊戲程式。物件導向程式設計會讓您能夠寫出使用者介面的相關應用，讓您輕鬆顯示和控制介面中相同類型的多個項目元素，這也是電玩遊戲程式中經常需要的功能。

物件導向程式設計是一種通用的開發風格，可用於程式開發的各個層面，遠遠超出我在本書開發電玩遊戲程式所示範的 OOP 技術。我由衷希望讀者覺得這種學習 OOP 的方式很有趣。

讓我們開始這趟學習的旅程吧！

1

程序式的 Python 程式範例

入門介紹性質的課程和書籍通常都會使用程序式程式設計風格來教授軟體開發，其中就有講述把程式拆分為多個函式（也有稱為程序或次程式）。您把資料傳遞給函式，而函式會執行一個或多個運算，然後傳回結果。

本書則介紹另一種不同的程式設計範式，稱之為**物件導向程式設計（OOP，object oriented programming）**，這種設計範式允許程式設計師以不同的方式來思考如何建構軟體。物件導向程式設計為程式設計師提供了一種把程式碼和資料組合成內聚單元的方法，從而避免了程序式程式設計中固有的一些複雜性問題。

我會透過建構兩個含有各種 Python 結構的小程式來回顧 Python 的一些基礎概念。第一個範例名稱為「Higher or Lower」的小型紙牌遊戲程式；第二個範例為模擬銀行的程式，對一個、兩個和多個帳戶執行銀行業務的相關操作。

兩者都會使用程序式程式設計的方法來建構，也就是使用資料和函式的標準技術來實作。隨後我還會利用 OOP 技術重寫這些程式。本章的目的是展示程序式程式設計中固有的一些關鍵問題，有了這種認知和理解後，接下來的章節會講解 OOP 是怎麼解決這些問題。

Higher or Lower 比大小紙牌遊戲

我的第一個範例是簡單的紙牌遊戲程式，名稱叫做 Higher or Lower（比大小遊戲）。在這個遊戲中，從一副牌中隨機挑選 8 張牌。第一張牌面朝上，遊戲要求玩家預測選中的下一張牌是否會比目前顯示的牌面「大（Higher）」或「小（Lower）」。舉例來說，假設顯示的牌面是 3，玩家選「大（Higher）」，然後翻開下一張牌。如果那張牌面的值真的比較大，則玩家猜對。在這個例子中，如果玩家選了「小（Lower）」，那就表示猜錯了。

如果玩家猜對了，就會得到 20 分，如果猜錯了，就扣 15 分。若下一張翻出的牌面與前一張的牌面大小相同，則算玩家猜錯。

表示資料

這支程式需要表示一副 52 張紙牌的資料，我會建構為一個串列（list）來進行處理。串列的 52 個元素中的每一個都是一個字典值（一組鍵/值對）。為了能表示任何一張紙牌，每個字典會有三個鍵/值對：牌名「rank」、花色「suit」和牌面值「value」。rank 是紙牌的名稱（Ace, 2, 3, ... 10, Jack, Queen, King），而 value 則是牌面的整數值（1, 2, 3, ... 10, 11, 12, 13 ）。例如，梅花 J（Jack of Clubs）會表示為以下字典：

```
{'rank': 'Jack', 'suit': 'Clubs', 'value': 11}
```

在玩家玩一輪之前，代表牌組的串列被建立並洗牌，讓牌的順序隨機化。我沒有讓紙牌以圖形來表示，所以每次使用者選擇「大（Higher）」或「小（Lower）」時，程式都會從牌組中取得牌的字典並為使用者印出 rank 和 suit，然後程式把新牌的牌面值與前一張牌的牌面值進行比較，並根據使用者猜答的正確性給出反饋。

實作

Listing 1-1 顯示了「Higher or Lower」遊戲的程式碼內容。

NOTE

提醒一下，本書中所有相關的程式碼檔案都可從 https://www.nostarch.com/object-orientated-python/ 或是作者的倉庫：https://github.com/IrvKalb/Object-Oriented-Python-Code/ 下載。您可以下載並執行範例程式碼，也可以自己練習輸入這些程式碼範例檔。

檔案：HigherOrLowerProcedural.py

Listing 1-1：使用程序式 Python 所編的 Higher or Lower 遊戲程式

```
#  HigherOrLower

import random

# 紙牌常數
SUIT_TUPLE = ('Spades', 'Hearts', 'Clubs', 'Diamonds')
RANK_TUPLE = ('Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen',
'King')

NCARDS = 8

# 傳入一個牌組，此函式會從牌組中返回一張隨機的牌
def getCard(deckListIn):
    thisCard = deckListIn.pop() # 從牌組頂端彈出一張牌並將其返回
    return thisCard

# 傳入一個牌組，此函式會返回牌組的洗牌後的副本
def shuffle(deckListIn):
    deckListOut = deckListIn.copy()  # 對起始牌組進行複製
    random.shuffle(deckListOut)
    return deckListOut

#  主程式
print('Welcome to Higher or Lower.')
print('You have to choose whether the next card to be shown will be higher or lower
than the current card.')
print('Getting it right adds 20 points; get it wrong and you lose 15 points.')
print('You have 50 points to start.')
print()

startingDeckList = []
for suit in SUIT_TUPLE: ❶
    for thisValue, rank in enumerate(RANK_TUPLE):
        cardDict = {'rank':rank, 'suit':suit, 'value':thisValue + 1}
        startingDeckList.append(cardDict)
```

```
score = 50

while True:  # 進行多次遊戲
    print()
    gameDeckList = shuffle(startingDeckList)
 ❷ currentCardDict = getCard(gameDeckList)
    currentCardRank = currentCardDict['rank']
    currentCardValue = currentCardDict['value']
    currentCardSuit = currentCardDict['suit']
    print('Starting card is:', currentCardRank + ' of ' + currentCardSuit)
    print()

 ❸ for cardNumber in range(0, NCARDS):   # 玩多張牌中的一場
        answer = input('Will the next card be higher or lower than the ' +
                               currentCardRank + ' of ' +
                               currentCardSuit + '?  (enter h or l): ')
        answer = answer.casefold()  # 強制轉成小寫字母
     ❹ nextCardDict = getCard(gameDeckList)
        nextCardRank = nextCardDict['rank']
        nextCardSuit = nextCardDict['suit']
        nextCardValue = nextCardDict['value']
        print('Next card is:', nextCardRank + ' of ' + nextCardSuit)

     ❺ if answer == 'h':
            if nextCardValue > currentCardValue:
                print('You got it right, it was higher')
                score = score + 20
            else:
                print('Sorry, it was not higher')
                score = score - 15

        elif answer == 'l':
            if nextCardValue < currentCardValue:
                score = score + 20
                print('You got it right, it was lower')

            else:
                score = score - 15
                print('Sorry, it was not lower')

        print('Your score is:', score)
        print()
        currentCardRank = nextCardRank
        currentCardValue = nextCardValue  # 不需要目前的花色

 ❻ goAgain = input('To play again, press ENTER, or "q" to quit: ')
    if goAgain == 'q':
        break

print('OK bye')
```

這支程式會先建立一個作為串列來放牌組❶。每張牌都是一個由 rank、suit 和 value 組成的字典。對於每一輪遊戲，我從牌堆中取出第一張牌並將其字典的元件儲存在變數中❷。對於接下來的 7 張牌，要求使用者猜測下一張牌比最近

出現的牌是大或是小❸。從牌堆中取出下一張牌，並將其字典的元件儲存到第二組變數中❹。遊戲把使用者的猜答與抽出的牌面值進行比較，並根據結果給予使用者反饋和分數❺。當使用者已經完成所有 7 張牌的猜答，我們會詢問是否要再次玩一次❻。

這支程式展示了許多程式設計的元素，特別是 Python 的變數、指定值陳述語、函式和函式呼叫、if/else 陳述句、print 陳述句、while 迴圈、串列、字串和字典等。本書是假設讀者對此範例中顯示的這些內容已有一定程度的熟悉。如果此程式中有任何您還不熟悉或不清楚的，那麼在繼續之前，您可能需要花一點時間查閱或複習相關的素材。

可重用程式碼

由於這是一款以紙牌為基礎的遊戲程式，因此程式碼顯然會建立和模擬一副紙牌的相關操作。如果我們想編寫另一種以紙牌為基礎的遊戲，能夠重用牌組和紙牌的程式碼會是很好的事情。

在程序式的程式中，通常很難完全識別出與程式某部分相關聯的所有程式碼片段，例如本範例中的牌組和紙牌的程式碼。在 Listing 1-1 中，牌組的程式碼由兩個元組常數、兩個函式、主程式中一些用來建構全域串列的相關程式碼所組成。全域串列中第一個是用來表示 52 張牌的起始牌組，而另一個全域串列則是用來表示正在玩的牌組。此外還請留意，就算在這樣的小程式中，資料和操作資料的程式碼也可能不會緊密地組合在一起。

因此，在另一支程式中重用牌組和紙牌的程式碼並不是那麼容易或直接。在第 12 章中，我們會重新審視這支程式，並展示 OOP 的解決方案，讓您了解以 OOP 開發的程式碼在重用時會怎麼變得更加容易的。

銀行帳戶模擬程式

在程序式程式碼的第二個範例中，我會展示模擬銀行營運的多個變體版本的程式碼，在程式的每個新版本中，我都會新增更多功能進去。請注意，這些程式並不符合真實上線執行，因為有無效的使用者輸入或誤用會造成錯誤。這裡程式的焦點放在讓您專注於程式碼與一個或多個銀行帳戶所關聯的資料進行互動交流。

首先是考量客戶想要對銀行帳戶進行哪些操作，以及代表的帳戶需要放入哪些資料。

分析所需的操作和資料

假設使用者想要對銀行帳戶進行的相關操作包括：

- 建立一個帳戶（create an account）
- 存款（deposit）
- 提款（withdraw）
- 查看餘額（check balance）

接下來是代表銀行帳戶所需的資料中至少要有：

- 顧客姓名
- 密碼
- 餘額

請留意，上述列出的所有操作都是動作（動詞），所有資料項目都是事物（名詞）。真實的銀行帳戶當然可以進行更多的操作，以及放入更多額外的資料（例如帳戶持有人的地址、電話和社會安全號碼），但為了讓書中的說明能更簡單清楚表達，我就以上述列出的 4 個動作和 3 個資料為起始。此外，為了保持範例的簡單和集中，所有金額的值都設為整數美元。另外還要提醒一點，在真實的銀行應用程式中，密碼不會像範例是以明文（未加密）的形式來儲存。

實作 1：單個帳戶且還沒有使用函式

在 Listing 1-2 一開始的版本中只有單個帳戶。

檔案：Bank1_OneAccount.py
Listing 1-2：單個帳戶的銀行模擬程式

```
# Non-OOP
# Bank Version 1
# 單個帳戶
```

```
accountName = 'Joe' ❶
accountBalance = 100
accountPassword = 'soup'

while True:
 ❷ print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press w to make a withdrawal')
    print('Press s to show the account')
    print('Press q to quit')
    print()

    action = input('What do you want to do? ')
    action = action.lower()  # 強制轉成小寫字母
    action = action[0]  # 只用第一個字母
    print()

    if action == 'b':
        print('Get Balance:')
        userPassword = input('Please enter the password: ')
        if userPassword != accountPassword:
            print('Incorrect password')
        else:
            print('Your balance is:', accountBalance)

    elif action == 'd':
        print('Deposit:')
        userDepositAmount = input('Please enter amount to deposit: ')
        userDepositAmount = int(userDepositAmount)
        userPassword = input('Please enter the password: ')

        if userDepositAmount < 0:
            print('You cannot deposit a negative amount!')

        elif userPassword != accountPassword:
            print('Incorrect password')

        else:  #OK
            accountBalance = accountBalance + userDepositAmount
            print('Your new balance is:', accountBalance)

    elif action == 's':  # 顯示
        print('Show:')
        print('       Name', accountName)
        print('       Balance:', accountBalance)
        print('       Password:', accountPassword)
        print()

    elif action == 'q':
        break

    elif action == 'w':
        print('Withdraw:')
```

```
        userWithdrawAmount = input('Please enter the amount to withdraw: ')
        userWithdrawAmount = int(userWithdrawAmount)
        userPassword = input('Please enter the password: ')

        if userWithdrawAmount < 0:
            print('You cannot withdraw a negative amount')

        elif userPassword != accountPassword:
            print('Incorrect password for this account')

        elif userWithdrawAmount > accountBalance:
            print('You cannot withdraw more than you have in your account')

        else:  #OK
            accountBalance = accountBalance - userWithdrawAmount
            print('Your new balance is:', accountBalance)

print('Done')
```

這支程式會先初始化三個變數來表示一個帳戶的資料❶。隨後顯示一個允許選擇操作的功能表❷。程式的主要程式碼直接作用於全域帳戶變數。

在這個例子中，所有的動作都放在一個主要層級中，程式碼中沒有函式。該程式執行良好，只是看起來可能有點長。想要讓較長的程式能更清晰且結構分明，其典型的做法是將某些功能相關的程式碼移到函式中，並在用到時呼叫這些函式。我們會在下一個實作銀行模擬程式的專案中對此進行探討。

實作 2：單個帳戶且有使用函式

在 Listing 1-3 的程式版本中，程式碼被分解為多個單獨的函式，每項功能對應一個函式。與前面程式相同的是這個模擬程式也只針對單個帳戶。

檔案：Bank2_OneAccountWithFunctions.py
Listing 1-3：單個帳戶且有函式的銀行模擬程式

```
# Non-OOP
# Bank 2
# 單個帳戶

accountName = ''
accountBalance = 0
accountPassword = ''

def newAccount(name, balance, password): ❶
    global accountName, accountBalance, accountPassword
    accountName = name
```

```
    accountBalance = balance
    accountPassword = password

def show():
    global accountName, accountBalance, accountPassword
    print('       Name', accountName)
    print('       Balance:', accountBalance)
    print('       Password:', accountPassword)
    print()

def getBalance(password): ❷
    global accountName, accountBalance, accountPassword
    if password != accountPassword:
        print('Incorrect password')
        return None
    return accountBalance

def deposit(amountToDeposit, password): ❸
    global accountName, accountBalance, accountPassword
    if amountToDeposit < 0:
        print('You cannot deposit a negative amount!')
        return None

    if password != accountPassword:
        print('Incorrect password')
        return None

    accountBalance = accountBalance + amountToDeposit
    return accountBalance

def withdraw(amountToWithdraw, password): ❹
 ❺ global accountName, accountBalance, accountPassword
    if amountToWithdraw < 0:
        print('You cannot withdraw a negative amount')
        return None

    if password != accountPassword:
        print('Incorrect password for this account')
        return None

    if amountToWithdraw > accountBalance:
        print('You cannot withdraw more than you have in your account')
        return None

 ❻ accountBalance = accountBalance - amountToWithdraw
    return accountBalance

newAccount("Joe", 100, 'soup')  # 建立一個帳戶

while True:
    print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press w to make a withdrawal')
    print('Press s to show the account')
```

```
    print('Press q to quit')
    print()

    action = input('What do you want to do? ')
    action = action.lower()  # 強制轉成小寫
    action = action[0]  # 只使用第一個字母
    print()

    if action == 'b':
        print('Get Balance:')
        userPassword = input('Please enter the password: ')
        theBalance = getBalance(userPassword)
        if theBalance is not None:
            print('Your balance is:', theBalance)

❼ elif action == 'd':
        print('Deposit:')
        userDepositAmount = input('Please enter amount to deposit: ')
        userDepositAmount = int(userDepositAmount)
        userPassword = input('Please enter the password: ')

    ❽ newBalance = deposit(userDepositAmount, userPassword)
        if newBalance is not None:
            print('Your new balance is:', newBalance)

---版面有限，省略呼叫對應函式的其他程式碼---

print('Done')
```

在這個版本中，我為銀行帳戶確定的每項操作（建立❶、檢查餘額❷、存款❸和提款❹）建構了一個函式，並重新排放了程式碼的位置，以便讓主程式部分可以取用呼叫對應其操作的函式。

從結果來看，主程式更具可讀性。舉例來說，如果使用者鍵入 d 表示他們想要存款❼，程式碼會呼叫名為 deposit() 的函式❸，傳入要存款的金額和使用者輸入的帳戶密碼。

不過，如果您查看任何一個函式的定義（例如，withdraw() 函式），您會看到程式碼是使用 global 陳述句❺來存取（取得或設定）代表帳戶的變數。在 Python 的語法中，只當您想更改函式中全域變數的值時才需要用到 global 陳述句。而我在這裡使用這種語法是為了更清楚地表明這些函式所參照指到的是全域變數，就算只是取得某個值也一樣。

以一般程式設計的原則來看，函式是不應該修改全域變數。函式應該只使用傳給它的資料，根據該資料進行運算，並返回一個或多個結果。在這支程式中的

withdraw() 函式確實能用，但是它違反了這個規則，它裡面修改了全域變數 accountBalance 的值❻（另外也存取了全域變數 accountPassword 的值）。

實作 3：兩個帳戶的版本

在 Listing 1-4 的程式版本中的銀行模擬程式使用與 Listing 1-3 相同的方法，但增加了處理兩個帳戶的能力。

檔案：Bank3_TwoAccounts.py
Listing 1-4：兩個帳戶且有函式的銀行模擬程式

```
# Non-OOP
# Bank 3
# 兩個帳戶

account0Name = ''
account0Balance = 0
account0Password = ''
account1Name = ''
account1Balance = 0
account1Password = ''
nAccounts = 0

def newAccount(accountNumber, name, balance, password):
 ❶ global account0Name, account0Balance, account0Password
    global account1Name, account1Balance, account1Password

    if accountNumber == 0:
        account0Name = name
        account0Balance = balance
        account0Password = password
    if accountNumber == 1:
        account1Name = name
        account1Balance = balance
        account1Password = password

def show():
 ❷ global account0Name, account0Balance, account0Password
    global account1Name, account1Balance, account1Password

    if account0Name != '':
        print('Account 0')
        print('       Name', account0Name)
        print('       Balance:', account0Balance)
        print('       Password:', account0Password)
        print()
    if account1Name != '':
        print('Account 1')
        print('       Name', account1Name)
```

```
        print('       Balance:', account1Balance)
        print('       Password:', account1Password)
        print()

def getBalance(accountNumber, password):
 ❸ global account0Name, account0Balance, account0Password
   global account1Name, account1Balance, account1Password

    if accountNumber == 0:
        if password != account0Password:
            print('Incorrect password')
            return None
        return account0Balance
    if accountNumber == 1:
        if password != account1Password:
            print('Incorrect password')
            return None
        return account1Balance

---以下省略其他 deposit() 和 withdraw() 函式的程式碼---

---以下省略主程式呼叫上述函式的程式碼---

print('Done')
```

就算只有兩個帳戶，您也會看到這種設計和編寫程式的方式很快就會失控。首先我們要為每個帳戶設定三個全域變數，分別在❶、❷和❸的位置。此外，每個函式現在都有一個 if 陳述句來判別要存取或更改哪一組全域變數。將來若是想要再添加另一個帳戶時，我們都需要在每個函式中添加另一組全域變數和更多的 if 陳述句來判定要處理哪個帳戶。這根本不是可行的開發設計方法。我們需要一種不同的方式來處理多個帳戶。

實作 4：使用串列的多個帳戶版本

為了能更輕鬆地容納多個帳戶，在 Listing 1-5 中，我會用串列來代表資料。我在這個版本的程式中會使用三個平行串列（parallel list）：accountNamesList、accountPasswordsList 和 accountBalancesList。

檔案：Bank4_N_Accounts.py
Listing 1-5：使用平行串列的銀行模擬程式

```
# Non-OOP Bank
# Version 4
# 多個帳戶：使用串列
```

```
accountNamesList = [] ❶
accountBalancesList = []
accountPasswordsList = []

def newAccount(name, balance, password):
    global accountNamesList, accountBalancesList, accountPasswordsList
    accountNamesList.append(name)
 ❷ accountBalancesList.append(balance)
    accountPasswordsList.append(password)

def show(accountNumber):
    global accountNamesList, accountBalancesList, accountPasswordsList
    print('Account', accountNumber)
    print('       Name', accountNamesList[accountNumber])
    print('       Balance:', accountBalancesList[accountNumber])
    print('       Password:', accountPasswordsList[accountNumber])
    print()

def getBalance(accountNumber, password):
    global accountNamesList, accountBalancesList, accountPasswordsList
    if password != accountPasswordsList[accountNumber]:
        print('Incorrect password')
        return None
    return accountBalancesList[accountNumber]

---版面有限，不印出其他函式---

# 建立兩個樣本帳戶
print("Joe's account is account number:", len(accountNamesList)) ❸
newAccount("Joe", 100, 'soup')

print("Mary's account is account number:", len(accountNamesList)) ❹
newAccount("Mary", 12345, 'nuts')

while True:
    print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press n to create a new account')
    print('Press w to make a withdrawal')
    print('Press s to show all accounts')
    print('Press q to quit')
    print()

    action = input('What do you want to do? ')
    action = action.lower()  # force lowercase
    action = action[0]  # just use first letter
    print()

    if action == 'b':
        print('Get Balance:')
     ❺ userAccountNumber = input('Please enter your account number: ')
        userAccountNumber = int(userAccountNumber)
        userPassword = input('Please enter the password: ')
```

```
        theBalance = getBalance(userAccountNumber, userPassword)
        if theBalance is not None:
            print('Your balance is:', theBalance)

---版面有限，不印出其他的使用者介面的程式碼---

print('Done')
```

在程式開始時，我把三個串列設定為空串列❶。將來在建立新帳戶時，我會把適當的值新增到三個串列中的每一個適當的位置❷。

由於現在的程式需要處理多個帳戶，因此我會使用最基本的銀行帳戶編號概念來處理。每次使用者建立帳戶時，程式碼都會對其中一個串列使用 len() 函式來取得長度值，並返回這個數字值當作使用者的帳戶編號❸❹。當我為第一位使用者建立帳戶時，accountNamesList 的長度為 0。因此，建立的第一個帳戶編號就會被指定為 0，第二個帳戶編號則指定為 1，依此類推。隨後就像在真實的銀行一樣，建立帳戶之後就可進行任何操作（如存款或提款），但使用者必須提供他們的帳戶編號❺。

不過在這裡的程式碼仍是處理全域資料，這個範例有三個全域資料的串列。

請想像一下，若以表格的形式來檢視這些資料，那排放的方式有可能類似於表 1-1 所示。

表 1-1：帳戶的資料表

Account number	Name	Password	Balance
0	Joe	**soup**	100
1	Mary	**nuts**	3550
2	Bill	**frisbee**	1000
3	Sue	**xxyyzz**	750
4	Henry	**PW**	10000

資料存放在三個全域的 Python 串列，其中每個串列代表此表中的一欄。舉例來說，正如從上表反白顯示的那一欄中所看到的那樣，所有密碼（password）值都被分組放在一個串列中。使用者的名稱（name）則被分組到在另一個串列內，而餘額（Balance）被分組到在第三個串列。使用這種方法後，若想要取得有關一個帳戶的全部資訊，您需要使用索引值分別存取這些串列中的值。

雖然這種處理方式可行，但似乎有點尷尬，資料並未按照正常邏輯來分組。舉例來說，把所有使用者的密碼放在一起似乎不太對。此外，每次想要在帳戶中加入新的屬性（例如地址或電話號碼）時，您都需要建立和存取其他全域串列才能配合。

相反地，您真正想要的存放方式應該像表 1-2 這樣，在表格中以橫向一列來進行分組。

表 1-2：帳戶的資料表

Account number	Name	Password	Balance
0	Joe	soup	100
1	Mary	nuts	3550
2	**Bill**	**frisbee**	**1000**
3	Sue	xxyyzz	750
4	Henry	PW	10000

使用這種方法來分析，每一列所代表的就是與單個銀行帳戶關聯的資料。雖然都是相同的資料，但這種分組方式能更自然地表示帳戶的內容。

實作 5：使用帳戶字典串列的版本

為了實作最後的版本，我會使用稍微複雜一點的資料結構。在這個版本中，我將建立一個帳戶串列，其中每個帳戶（串列的每個元素）都是一個字典，如下所示：

```
{'name':<someName>, 'password':<somePassword>, 'balance':<someBalance>}
```

NOTE

在本書中的程式碼內，每當我在尖括號（<>）中表示某個值時，這代表著您應該改用您選擇的值來替換該項目（包括括號）。舉例來說，在前面的程式碼行中，<someName>、<somePassword> 和 <someBalance> 是佔位符號，應讓替換成您使用的實際值。

最後實作版本的程式碼展示在 Listing 1-6 中。

檔案：Bank5_Dictionary.py
Listing 1-6：使用字典串列的銀行模擬程式

```
# Non-OOP Bank
# Version 5
# 多個帳戶：使用字典串列

accountsList = [] ❶

def newAccount(aName, aBalance, aPassword):
    global accountsList
    newAccountDict = {'name':aName, 'balance':aBalance, 'password':aPassword}
    accountsList.append(newAccountDict) ❷

def show(accountNumber):
    global accountsList
    print('Account', accountNumber)
    thisAccountDict = accountsList[accountNumber]
    print('       Name', thisAccountDict['name'])
    print('       Balance:', thisAccountDict['balance'])
    print('       Password:', thisAccountDict['password'])
    print()

def getBalance(accountNumber, password):
    global accountsList
    thisAccountDict = accountsList[accountNumber] ❸
    if password != thisAccountDict['password']:
        print('Incorrect password')
        return None
    return thisAccountDict['balance']

---版面有限，不印出 deposit() 和 withdraw() 函式---

# 建立二個樣本帳戶
print("Joe's account is account number:", len(accountsList))
newAccount("Joe", 100, 'soup')

print("Mary's account is account number:", len(accountsList))
newAccount("Mary", 12345, 'nuts')

while True:
    print()
    print('Press b to get the balance')
    print('Press d to make a deposit')
    print('Press n to create a new account')
    print('Press w to make a withdrawal')
    print('Press s to show all accounts')
    print('Press q to quit')
    print()

    action = input('What do you want to do? ')
    action = action.lower()  # 強制轉成字寫
    action = action[0]  # 只用第一個字母
    print()
```

```
    if action == 'b':
        print('Get Balance:')
        userAccountNumber = input('Please enter your account number: ')
        userAccountNumber = int(userAccountNumber)
        userPassword = input('Please enter the password: ')
        theBalance = getBalance(userAccountNumber, userPassword)
        if theBalance is not None:
            print('Your balance is:', theBalance)

    elif action == 'd':
        print('Deposit:')
        userAccountNumber= input('Please enter the account number: ')
        userAccountNumber = int(userAccountNumber)
        userDepositAmount = input('Please enter amount to deposit: ')
        userDepositAmount = int(userDepositAmount)
        userPassword = input('Please enter the password: ')

        newBalance = deposit(userAccountNumber, userDepositAmount, userPassword)
        if newBalance is not None:
            print('Your new balance is:', newBalance)

    elif action == 'n':
        print('New Account:')
        userName = input('What is your name? ')
        userStartingAmount = input('What is the amount of your initial deposit? ')
        userStartingAmount = int(userStartingAmount)
        userPassword = input('What password would you like to use for this account? ')

        userAccountNumber = len(accountsList)
        newAccount(userName, userStartingAmount, userPassword)
        print('Your new account number is:', userAccountNumber)

---版面有限，不印出使用者介面的程式碼---

print('Done')
```

使用這種方法就可以在單個字典中找到與一個帳戶相關聯的所有資料❶。若想要建立一個新帳戶，我們建構一個字典並將其新增到帳戶串列❷即可。每個帳戶都會分配一個數字編號（一個簡單的整數值），在對該帳戶執行任何操作時必須提供這個帳戶編號。舉例來說，使用者在存款時要提供帳戶的編號，get Balance() 函式會使用該編號作為帳戶串列的索引❸。

這樣的做法會清理很多東西，讓資料的組織更加合乎邏輯。但程式中的每個函式仍然必須去存取全域的帳戶串列。正如我們會在下一節中學到的，授予函式存取所有帳戶資料的權限會帶來潛在的安全風險。在理想情況下，每個函式應該只能影響單個帳戶的資料。

程序式實作的常見問題

本章中的範例程式都有一個共同的問題：函式操作的所有資料都儲存在一個或多個全域變數中。由於以下原因，在程序式程式設計中使用大量的全域資料是不好的設計習慣：

1. 任何使用和（或）更改全域資料的函式都不能輕易地在不同的程式中重用。存取全域資料的函式是對放在與函式本身程式碼不同（更高）層級的資料進行操作，該函式會需要一個 global 陳述句來存取這些資料。您無法直接取用一個依賴於全域資料的函式並在另一程式中重用。這樣的函式只能在具有相似全域資料的程式中重複使用。

2. 許多程序式的程式往往有大量的全域變數。根據定義，全域變數可以被程式中任何地方的任何程式碼使用或更改。對全域變數指定值的寫法通常會廣泛分佈在整個程序式的程式中，包括在主程式和函式內部。因為變數的值會在任何地方發生變化，以這種方式來編寫的程式在除錯和維護都非常困難。

3. 為了使用全域資料而設計編寫的函式通常會存取超過本身需要的資料內容。當函式使用全域串列、字典或任何其他全域資料結構時，它能存取該資料結構中的所有資料，但函式應該只需要對該資料的某部分（或少量）進行操作。能夠讀取和修改大型資料結構中的所有資料可能會造成錯誤，例如意外使用或覆蓋了函式不打算接觸的某些資料。

物件導向的解決方案：第一個類別

Listing 1-7 是一種物件導向的開發方法，它把單個帳戶的所有程式碼和相關資料結合起來。這裡有很多新的概念觀點，我會從下一章開始詳細介紹這些新的概念。目前我並不希望您能完全理解這個範例程式，但請留意，在這個腳本（稱為**類別**）中有程式碼和資料的結合成果。這是您在本書中看到的第一個物件導向程式碼。

檔案：Account.py

Listing 1-7：第一個以 Python 編寫的類別範例程式碼

```
# Account class

class Account():
    def __init__(self, name, balance, password):
        self.name = name
        self.balance = int(balance)
        self.password = password

    def deposit(self, amountToDeposit, password):
        if password != self.password:
            print('Sorry, incorrect password')
            return None

        if amountToDeposit < 0:
            print('You cannot deposit a negative amount')
            return None

        self.balance = self.balance + amountToDeposit
        return self.balance

    def withdraw(self, amountToWithdraw, password):
        if password != self.password:
            print('Incorrect password for this account')
            return None

        if amountToWithdraw < 0:
            print('You cannot withdraw a negative amount')
            return None

        if amountToWithdraw > self.balance:
            print('You cannot withdraw more than you have in your account')
            return None

        self.balance = self.balance - amountToWithdraw
        return self.balance

    def getBalance(self, password):
        if password != self.password:
            print('Sorry, incorrect password')
            return None
        return self.balance

    # 用來除錯所顯示的內容
    def show(self):
        print('       Name:', self.name)
        print('       Balance:', self.balance)
        print('       Password:', self.password)
        print()
```

現在請看一下這些函式，看看它們與之前的程序式的程式範例有何相似之處。這些函式的名稱與前面程式碼中所用的大致相同：show()、getBalance()、deposit() 和 withdraw()，但您還會看到 self（或 self.）這個字貫穿整支程式碼，您會在本書接下來的章節中了解這個字所代表的什麼意思。

總結

本章是從名為「Higher or Lower」比大小紙牌遊戲程式的實作開始。在第 12 章中我會展示如何製作這個紙牌遊戲的圖形使用介面物件導向版本。

接著介紹了模擬銀行相關操作的程式，從單個帳戶到多個帳戶的版本。這裡討論了使用程序式實作所模擬的幾種不同方法，並描述了這些方法所產生的一些問題。最後則是展示第一個以物件導向方法所編寫的類別，以這個類別來模擬銀行帳戶和相關操作。