前言

感謝你購買本書。Python 是這世界最受歡迎的程式語言，而且有各式各樣背景的人變成 Python 程式設計師；其中有些擁有正式的電腦科學教育背景，有些則是因為興趣而學習 Python，更有一些雖然是在專業環境使用 Python，但他們主要的工作並非軟體開發人員。這本中階書籍裡的問題將有助於老鳥程式設計師釐清他們在接受電腦科學教育時，學習程式語言的某些進階功能的想法；對自學的程式設計師來說，以他們所選擇的程式語言（Python）來學習經典問題，則能加快他們電腦科學教育的學習速度。這本書涵蓋了電腦科學領域的諸多解決問題的技巧，相信每個人都能找到感興趣的主題。

這本不是 *Python* 入門書，反之，這本書假設你已經是中上程度的 Python 程式設計者。雖然本書要求的是 Python 3.7，但並未假設讀者能掌控這個最新版本的每個面向；事實上，創作此書的前提是書中內容能作為學習材料來幫助讀者完成如此的掌控。此外，這本書並不適合完全沒有接觸過 Python 的讀者。

為什麼選 Python ？

很多研究或工作都用到 Python，諸如資料科學、影片製作、電腦科學教育、資訊管理等。的確沒有 Python 沒有涉及的運算領域（核心開發可能是例外），這個程式語言因為彈性、優美和簡潔的語法、純正的物件導向和活躍的社群而受到關
愛。強而有力的社群至關重要，因為它意味著 Python 歡迎新手，並且提供了大量可用的程式庫生態系統給開發人員。

因為這樣，有時候會認為 Python 是適合初學者的程式語言，而且這種特徵可能也很正確。例如多數人皆同意 Python 比 C++ 容易學習，而且幾乎可以確定 Python 社群對新手更為友善。所以很多人學習 Python 是因為 Python 平易近人，接著他們很快就開始編寫心中想要的程式。不過他們可能從未受過電腦科學教育，也就是完全沒有學過這些十分有用的問題解決技巧。如果你就是這類已經瞭解 Python 但還不瞭解電腦科學的程式設計師，那麼你正是本書的目標讀者。

其他人在長期的軟體開發之後，會將 Python 當作第 2、第 3、第 4 或第 5 種語言而加以學習。對他們來說，看到這些他們已經在其他語言看過的舊問題，也有助於他們加速學習 Python。對他們來說，這本書可能是工作面試之前很好的複習，或者也可能發現一些先前未曾想過可應用在工作上的問題解決技巧。若是這類的讀者，我鼓勵他們瀏覽目錄來找找本書裡面有沒有能引起他們興趣的主題。

何謂經典電腦科學問題？

有人說電腦與電腦科學的關係就如同望遠鏡與天文學，如果真是這樣，那或許程式語言就像是望遠鏡的鏡片。無論如何，這裡所謂的「經典電腦科學問題」，是指「通常會在大學電腦科學課程教授的程式設計問題」。

新手程式設計師必須要能解決一些程式設計問題，這類問題不論是在攻讀學士學位（電腦科學、軟體工程等）的課堂裡，或是中階程式設計教科書當中（例如人工智慧或演算法的第 1 本書），都是老生常談，因而視為經典。你會在這本書找到精心挑選過的這類問題。

這些問題的難易程度可以小到幾行程式碼就能解決，也能複雜到需要花好幾章節來組構系統才能處理。有些問題涉及人工智慧，有些則只需要常識。有些問題很實用，但也不乏稀奇古怪的問題。

什麼類型的問題會出現在這本書？

第 1 章簡介了多數讀者可能很熟悉的問題解決技巧，諸如遞迴、備忘、位元操作都是即將在後續章節探索的其他技巧的重要基礎。

跟在這個四平八穩的簡介之後，是將重點放在搜尋問題的第 2 章。搜尋是個很大的議題，這本書大多數的問題大概都可以納入搜尋的麾下。第 2 章介紹了最重要的搜尋演算法，包括二分搜尋、深度優先搜尋、寬度優先搜尋、A*。這些演算法還會在本書後續再次用到。

在第 3 章，你會為了要解決各種問題而建置應用框架，這類的問題可以藉由彼此相互約束的有限值域變數而抽象定義，包括經典的八皇后問題、澳洲地圖著色問題、密碼算術 SEND+MORE=MONEY。

第 4 章探索了圖形演算法的世界；對初學者來說，圖形演算法可以應用的範圍是讓人驚訝的寬廣。你將在本章建置圖形資料結構，再用它們來解決數種經典的最佳化問題。

第 5 章將探索基因演算法，這項技術的重要性雖然低於本書所涵蓋的其他技術，但有時可以解決傳統演算法無法在合理時間範圍內解決的問題。

第 6 章涵蓋了 k-means 群聚演算法，而這或許是本書演算法最為具體的章節。這種分類群聚的技術容易實作、易於瞭解，而且應用廣泛。

第 7 章的目的在解說何謂類神經網路，並且提供最簡單的類神經網路讓讀者淺嚐。本章的目的並非完整說明這個讓人興奮且持續發展的領域。你將在本章以第一原理建置類神經網路，而且不使用外部程式庫，因此能真的瞭解類神經網路的運作方式。

第 8 章的內容是在雙人完美資訊賽局進行對抗式搜尋。你將學習一種稱為 minimax 的搜尋演算法，這種演算法可以用來開發諸如西洋棋、西洋跳棋和四子棋等遊戲的人造對手。

最後的第 9 章涵蓋了有趣（也好玩）的問題，這些問題不適合放在本書的其他位置。

這本書為誰而寫？

這本書寫給中階和老鳥程式設計人員。如果是想要深入 Python 知識的老鳥程式設計人員，會從這裡找到和他們電腦科學教育或程式設計教育相當熟悉的問題。若是中階程式設計人員，則會以他們選擇的語言（Python）向他們介紹這些經典問題。準備要面試的開發人員可能會發現這本書是很有價值的準備素材。

除了專業的程式設計師，對 Python 有興趣的大學電腦科學課程的學生可能會發現這本書很有幫助。它並不打算以嚴謹的態度來介紹資料結構和演算法。**這不是資料結構和演算法的教科書**。你在此書頁面看不到大 O 記法的驗證，反之，這是一本容易閱讀的問題解決技巧，而且這些技巧應該是採用資料結構、演算法和人工智慧類別的完成品。

再次強調，本書假設讀者已經具備 Python 語法和語意的知識。沒有程式設計經驗的讀者無法從本書獲得太多，而如果沒有 Python 經驗的話，幾乎可以肯定會非常掙扎。此外，《經典電腦科學問題解析—使用 *Python*》是一本寫給將 Python 運用在工作的程式設計師、以及電腦科學學生的書。

Python 版本、原始碼儲藏庫及型別提示

這本書的原始碼恪遵 3.7 版的 Python 語言編寫，並且用了 Python 3.7 獨有的功能，所以部分程式碼將無法在先前的 Python 版本執行。開始本書之前，請務必下載最新版本的 Python，而不是辛苦的試著要在舊版 Python 執行這些範例。

這本書只使用 Python 標準程式庫（第 2 章稍有例外，因為安裝了 `typing_extensions` 模組），所以本書所有的程式碼應該能在任何支援 Python 的平台執行（macOS、Windows、GNU/Linux 等）。本書程式碼只針對 CPython 測試過（可以從 python.org 取得的主要 Python 直譯器），但大部分的程式碼可以在 Python 3.7（另一個 Python 直譯器）執行。

這本書不會解釋如何使用諸如編輯器、除錯器及 Python REPL 之類的 Python 工具，書中的程式碼可以從 GitHub 儲藏庫取得：https://github.com/davecom/ClassicComputerScienceProblemsInPython；這些程式碼按照章節放進不同的資料夾。本書會在列出每段程式碼的標頭呈現原始檔名稱，你可以在 GitHub 儲藏庫對應的章節資料夾找到那些原始檔。根據你電腦的 Python 3 直譯器名稱的設定，只要輸入 `python3 filename.py` 或 `python filename.py`，應該就能執行程式。

列在本書的所有程式碼都使用了 Python 型別提示，也就是所謂的型別註解。這些註解是 Python 語言相對較新的功能，這項功能看起來可能會讓以前從未看過的 Python 程式設計人員感到害怕。本書之所以使用是因為以下 3 點理由：

1. 它們能讓變數、函式參數、函式傳回值的型別更清楚。
2. 因為第 1 點理由，使用型別提示能讓程式碼有某種程度的自我陳述。你只需檢視它的簽名碼，而不用從註解或陳述字串找出函式的傳回值型別。
3. 為了確保正確，它們允許對程式碼進行型別檢查，mypy 就是其中一種很受歡迎的 Python 型別檢查程式。

並非人人都是型別提示的粉絲，而且整本書都用了型別提示，老實說相當冒險。我希望它們是助力而非阻力。以型別提示編寫 Python 需要一點時間，但卻能讓你

回頭讀到更清楚的程式碼。有趣的是型別提示對 Python 直譯器實際執行程式碼並沒有影響。就算你移除本書任何程式碼裡的型別提示，程式碼依然可以執行。如果你之前從未看過型別提示，而且在深入本書之前自認需要更全面的型別提示介紹，請參閱附錄 C，那裡快速介紹了型別提示。

沒有圖形、沒有 UI 程式碼，只有標準程式庫

本書沒有任何範例的執行結果會產生圖形，也沒有任何範例會使用圖形使用者介面（GUI）。為什麼？因為我的目標是盡可能使用簡潔和清楚易讀的方法來解決相關的問題。通常，製作圖形會妨礙甚或絕對會使解決方法比需要說明的技巧或演算法更複雜。

再者，因為不使用任何 GUI 應用框架，本書所有程式碼的可攜性都非常高，在 Linux 執行 Python 嵌入式版本就像在 Windows 桌面執行一樣容易。此外，自行決定只使用 Python 標準程式庫（而非任何外部程式庫）裡的套件，就像大多數高階 Python 書籍的作法。為什麼？目標是從第一原理所教授的問題解決技巧，而不是「嗶嗶嗶安裝解決方案」，然後結束。藉由從零開始解決每個問題，你有希望能瞭解廣為普及流行的程式庫在幕後的運作方式。至少，只使用標準程式庫會提高本書程式碼的可攜性，並且易於執行。

這不是說圖形解決方案對演算法的解說能力比不上非圖形的解決方案，這根本不是本書的重點，而是圖形會增加另一層不必要的複雜性。

1 熱身的小問題

一開始，我們將探討一些只需幾個相對較短的函式就能解決的簡單問題。雖然這些只是小問題，但它們依然能讓我們探討一些解決問題的有趣技巧。請把這些問題視為很好的熱身。

1.1 費式數列

費氏數列是一組數列，這組數列除了第 1 和第 2 項之外，其他任何數的值皆為前兩項的總和：

```
0, 1, 1, 2, 3, 5, 8, 13, 21...
```

費氏數列的第 1 個數值是 `0`，費氏數列的第 4 個數值是 `2`，以此類推，若要得到數列裡的任何費氏數值 `n`，可以利用這項公式

```
fib(n) = fib(n - 1) + fib(n - 2)
```

1.1.1 初試遞迴

上述計算費氏數值（如圖 1.1 所示）的公式是虛擬碼的形式，可以簡單改寫成**遞迴** Python 函式（遞迴函式是一種自我呼叫的函式）。這種機械化的改寫將充當我們第一次嘗試編寫函式，而所編寫的這個函式會傳回特定費氏數列的值。

程式 1.1　fib1.py

```
def fib1(n: int) -> int:
    return fib1(n - 1) + fib1(n - 2)
```

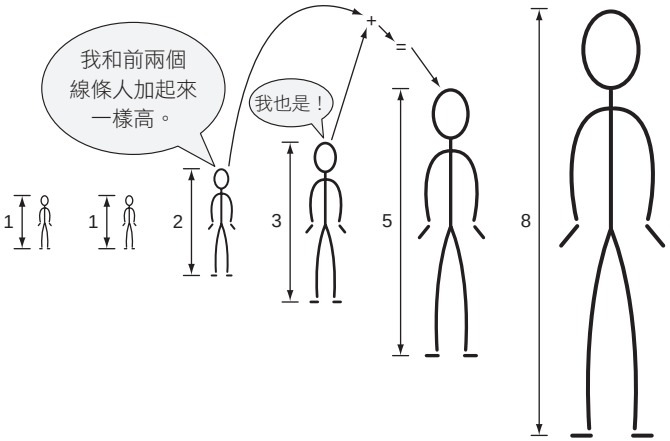

圖 1.1 每個線條人的高度是前兩個線條人高度的加總。

讓我們試著以某個值來呼叫這個函式，並以此執行。

程式 1.2 fib1.py 承上

```
if __name__ == "__main__":
    print(fib1(5))
```

喔哦！如果我們試著執行 fib1.py，就會發生錯誤：

```
RecursionError: maximum recursion depth exceeded
```

問題在於 `fib1()` 將會永遠一直執行，而且不會傳回最終結果。每次呼叫 `fib1()` 就會再另外呼叫兩次 `fib1()` 而無止盡看不到盡頭。我們將這種情況稱為**無窮遞迴**（如圖 1.2），而它類似**無窮迴圈**。

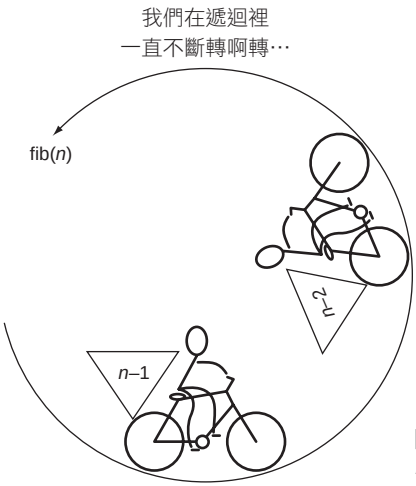

圖 1.2　遞迴函式 **fib(n)** 以引數 **n-2** 和 **n-1** 自我呼叫。

1.1.2　利用基本情況

請注意，一直到執行 `fib1()` 之前，你的 Python 環境都沒有任何跡象顯示其中有任何問題。避免無窮遞迴是程式設計人員的責任，而不是編譯器或直譯器的責任。無窮遞迴的原因是我們從未指定基本情況。遞迴函式裡的基本情況（base case）是作為停止點。

費氏數列函式這個例子的前兩個特殊序列值 `0` 和 `1`，就是天生自然的基本情況。不論 `0` 或 `1` 都不是數列裡前兩個數值的和；反之，它們是前兩個特殊的值。讓我們試著將它們指定成基本情況。

程式 1.3　fib2.py

```
def fib2(n: int) -> int:
    if n < 2:  # 基本情況
        return n
    return fib2(n - 2) + fib2(n - 1)  # 遞迴情況
```

NOTE　一如我們原本的想法，`fib2()` 版的費氏數列函式傳回 `0`，表示是第 0 個數值（`fib2(0)`），而非第 1 個數值。就程式設計而言，這種作法很有意義，因為我們習慣從第 0 個元素開始的順序。

`fib2()` 可以成功呼叫，而且也能傳回正確的值。以下試著以一些較小的數值來呼叫它。

程式 1.4 fib2.py 承上

```
if __name__ == "__main__":
    print(fib2(5))
    print(fib2(10))
```

請勿嘗試呼叫 `fib2(50)`，這將會永不停止的一直執行！為什麼？因為每次呼叫 `fib2()` 就會經由遞迴呼叫 `fib2(n - 1)` 和 `fib2(n - 2)` 而導致再次呼叫 `fib2()`（如圖 1.3）。也就是說，呼叫的樹狀是以指數成長。舉例來說，呼叫 `fib2(4)` 會導致這整組的呼叫：

```
fib2(4) -> fib2(3), ib2(2)
fib2(3) -> fib2(2), ib2(1)
fib2(2) -> fib2(1), ib2(0)
fib2(2) -> fib2(1), ib2(0)
fib2(1) -> 1
fib2(1) -> 1
fib2(1) -> 1
fib2(0) -> 0
fib2(0) -> 0
```

如果計算它們的次數（就如稍後會看到加入一些列印的呼叫），只運算到第 4 個元素就呼叫了 9 次 `fib2()`！更糟的是呼叫了 15 次，只是為了運算元素 5，而運算元素 10 需要 177 次呼叫、運算元素 20 需要 21,891 次呼叫。我們可以做得更好。

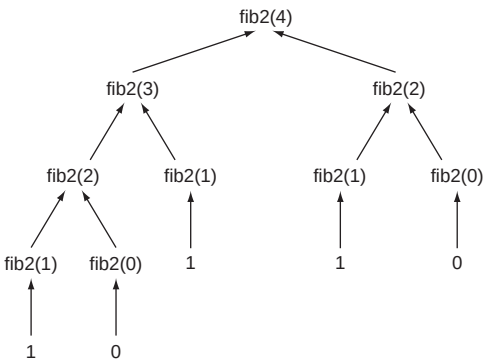

圖 1.3 每次非基本情況的呼叫，都會導致呼叫兩次 **`fib2()`**。

1.1.3 備忘法上場救援

備忘法是一種技巧，當你完成運算工作時便先儲存結果，因此若再次需要它們，就可以將它們找出來直接使用，無須再花 1 秒鐘的時間來計算它們（如圖 1.4）。[1]

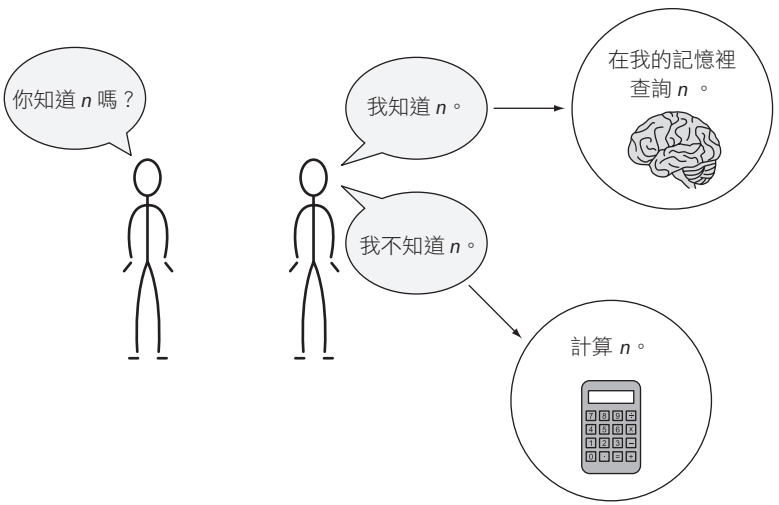

圖 1.4 人類的備忘機制。

讓我們另寫一個費氏數列函式，這個新版本函式會利用 Python 的字典功能來處理備忘法。

程式 1.5 fib3.py

```
from typing import Dict
memo: Dict[int, int] = {0: 0, 1: 1}  # 我們的基本情況

def fib3(n: int) -> int:
    if n not in memo:
        memo[n] = fib3(n - 1) + fib3(n - 2)  # 備忘法
    return memo[n]
```

現在呼叫 `fib3(50)` 就安全了。

1 備忘法（*memoization*）是由知名的英國電腦科學家 Donald Michie 所創。Donald Michie, *Memo functions: a language feature with "rote-learning" properties*（愛丁堡大學，機器智能與感知學系，1967。）

程式 1.6 fib3.py 承上

```
if __name__ == "__main__":
    print(fib3(5))
    print(fib3(50))
```

相對於呼叫 fib2(20) 會產生 21,891 次的 fib2() 呼叫，呼叫 fib3(20) 將只會呼叫 39 次的 fib3()。我們預先在 memo 填入了前面的基本情況 0 和 1，省下在 fib3() 再加入一段 if 陳述式的麻煩。

1.1.4 備忘法自動化

我們可以進一步簡化 fib3()，因為 Python 內建了能自動備忘任何函式的修飾器。在 fib4() 裡面搭配修飾器 @functools.lru_cache() 使用的程式碼，和我們在 fib2() 裡使用的完全相同。每次以新的引數執行 fib4() 時，修飾器就會將傳回值存入快取，當後續再以相同的引數呼叫 fib4() 時，就會從快取取出該引數的 fib4() 傳回值，並且傳回。

程式 1.7 fib4.py

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fib4(n: int) -> int:  # 和 fib2( ) 的定義相同
    if n < 2:  # 基本情況
        return n
    return fib4(n - 2) + fib4(n - 1)  # 遞迴情況

if __name__ == "__main__":
    print(fib4(5))
    print(fib4(50))
```

請注意，即使費氏數列函式的主體和 fib2() 裡的相同，我們還是可以馬上計算 fib4(50)。@lru_cache 的 maxsize 屬性表示它所修飾的函式最多應該要快取儲存的呼叫次數，若設為 None 表示沒有限制。

1.1.5 簡單的費氏數列

除此之外，還有效能更好的選擇，也就是以老派的迭代作法來解決費氏數列。

程式 1.8 fib5.py

```
def fib5(n: int) -> int:
    if n == 0: return n  # 特殊情況
    last: int = 0  # 初始設定成 fib(0)
    next: int = 1  # 初始設定成 fib(1)
    for _ in range(1, n):
        last, next = next, last + next
    return next

if __name__ == "__main__":
    print(fib5(5))
    print(fib5(50))
```

WARNING fib5() 裡的 for 迴圈主體使用了多元組這種或許有點過於巧妙的方式來還原；有些人可能覺得這犧牲了簡潔的可讀性，另外有些人則可能發現簡潔本身更具可讀性。重點是將 last 設成 next 之前的值，並且將 last 之前的值和 next 之前的值相加後指定給 next。這樣可以避免在 last 更新之後、next 更新之前建立暫時的變數來保存 next 的舊值。在 Python 應用多元組還原於某些類型的變數交換，是很常見的作法。

利用這種方法，for 迴圈的主體最多將執行 n - 1 次；也就是說，這仍然是效率最高的版本。比較執行 19 次 for 迴圈主體和 21,891 次 fib2() 遞迴呼叫的第 20 個費氏數列的數值，可能會對實際的應用程式產生重大影響！

在遞迴解決方式由後往前算，在迭代解決方式則是由前往後。有時候遞迴是解決問題最直覺的作法。舉例來說，fib1() 和 fib2() 的內容幾乎就是原始費氏數列公式的無意識機械翻譯。然而，天真幼稚的遞迴解決方式也可能帶來明顯的效能成本。請記住，任何能以遞迴解決的問題，都能以迭代解決。

1.1.6 以產生器產生費氏數列

截至目前，我們編寫了能產生費氏數列單一數值的函式，但如果我們想要產生直到某個數值的整組數列呢？其實使用 yield 陳述式就很容易將 fib5() 轉換成 Python 產生器。當產生器迭代時，每次迭代都將使用 yield 陳述式從費氏數列吐出 1 個值。

程式 1.9 fib6.py

```
from typing import Generator

def fib6(n: int) -> Generator[int, None, None]:
    yield 0  # 特殊情況
    if n > 0: yield 1  # 特殊情況
    last: int = 0  # 初始設定成 fib(0)
    next: int = 1  # 初始設定成 fib(1)
    for _ in range(1, n):
        last, next = next, last + next
        yield next  # 主要產生步驟

if __name__ == "__main__":
    for i in fib6(50):
        print(i)
```

如果執行 fib6.py，就會看到 51 個印出來的費氏數列的值。每次 for 迴圈迭代的 `for i in fib6(50):`，`fib6()` 會透過 yield 陳述式執行。如果到了函式盡頭而且也沒有 yield 陳述式了，迴圈就會結束迭代。

1.2 微不足道的壓縮

節省空間（虛擬或真實）通常很重要。使用更少的空間會更有效率，而且還可以省錢。如果你租的房子大於物品和人員的需要，就可以「縮小」到較便宜的小地方。如果伺服器所儲存的資料是按照位元組的多寡付費，你可能會想壓縮那些資料來減少儲存成本。**壓縮**是取得資料並加以編碼（改變其形式）的動作，這樣的方式可以讓資料佔用較少的空間。**解壓縮**則是反向的過程，將資料還原成原本的形式。

如果壓縮資料能有更高的儲存效率，那為何不壓縮所有的資料？因為要在時間和空間之間折衷。壓縮資料以及將它解壓縮回原本的形式需要時間。因此，資料壓縮唯一有意義的情況是優先考量資料縮小，而非加快執行。想想透過網際網路傳送的大型檔案，壓縮它們是有意義的，因為傳送檔案所花的時間比收到檔案然後解壓縮的時間還要久。此外，為了在原本的伺服器儲存而壓縮檔案所花的時間只僅需要考慮一次。

當你意識到資料儲存類型使用的位元，比嚴格要求的內容更多的時候，最簡單的資料壓縮就會出現。例如，想想低階的情況，如果將永遠不會超過 65,535 的無號整數儲存成 64 位元無號整數，就會導致儲存效率不佳。如果改成 16 位元無號整數來儲存，實際的空間消耗就能減少 75%。倘若有數百萬這樣的儲存效率不佳的數字，浪費的空間可能高達數百萬位元組（megabytes）。

有時為了簡單起見（當然這是正當的目標），Python 開發人員可以不用思考太多細節。Python 沒有 64 位元無號整數型別，也沒有 16 位元無號整數型別，只有 1 種可以儲存任意精確度的 `int` 型別；而函式 `sys.getsizeof()` 則有助你找出你的 Python 專案消耗了多少記憶體位元組。但因為 Python 物件系統本來就有固定負荷，所以無法在 Python 3.7 建立佔用不到 28 個位元組（224 個位元）的 `int`。單一個 `int` 可以一次擴展 1 個位元（就如我們即將在這個範例所示範的），但最少會耗用 28 個位元組。

> **NOTE** 如果對二進位有點生疏，請回想一下，位元就是非 1 即 0 的單一值，然後再以一連串基底為 2 的 0、1 數列來表示數值。對這一節的目的而言，你不需要執行任何基底為 2 的數學運算，但需要瞭解儲存型別的位元數量決定了它可以表示多少不同的值。舉例來說，1 個位元可以表示 2 個值（0 或 1），2 個位元可以表示 4 個值（00、01、10、11），3 個位元可以表示 8 個值，依此類推。

如果型別打算用型別表示的不同數值的數量，小於用來儲存它能表示的位元的數值數量，很可能就可以更有效率的儲存。我們以 DNA[2] 裡形成基因的核苷酸為例：每個核苷酸只能是 A、C、G、T 等 4 個值的其中一個（第 2 章將會有更多相關內容），但若將基因儲存成 `str`（可以想成是 Unicode 字元集合），每個核苷酸將會由一個字元表示，這通常需要 8 個位元的儲存空間。若以二進制儲存這 4 種可能的值，則只需要 2 個位元來儲存，也就是以 00、01、10、11 來表示這 4 個不同的值，並且以 2 個位元來儲存。如果將 00 指定成 A、01 指定成 C、10 指定成 G、11 指定成 T，核苷酸字串所需要的儲存空間就可以減少 75%（每個核苷酸從 8 個位元降到 2 個位元）。

2 這個範例的靈感來自 Robert Sedgewick 和 Kevin Wayne 著作的《*Algorithms*》第 4 版（Addison-Wesley Professional, 2011）的第 819 頁。

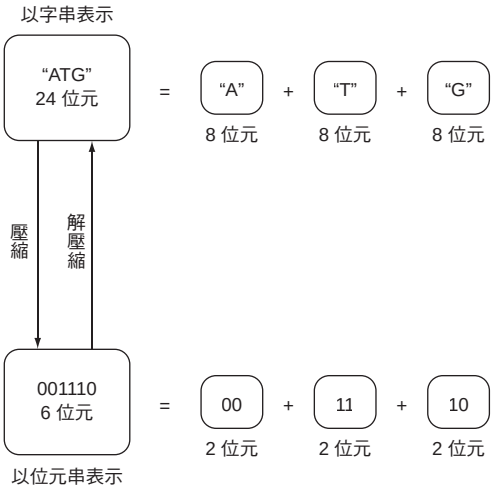

圖 1.5 將表示 1 個基因的 str 壓縮成每個核苷酸 2 位元的位元串。

也就是不將核苷酸儲存成為 `str`，而是儲存成**位元串**（如圖 1.5）。位元串就如同它聽起來的樣子：諸多 1 和 0 的任意長度數列。但偏偏 Python 標準程式庫並不包含處理任意長度位元串的現成構造。以下的程式碼會將諸多 A、C、G、T 組成的 `str` 轉換成位元串，然後再轉換回來。這個位元串儲存在 `int` 裡。因為 Python 的 `int` 型別可以是任何長度，也可以當作任何長度的位元串來用。為了要轉換回 `str`，我們將會實作 Python 的 `__str__()` 特殊方法。

程式 1.10 trivial_compression.py

```
class CompressedGene:
    def __init__(self, gene: str) -> None:
        self._compress(gene)
```

`CompressedGene` 提供了基因裡表示核苷酸的 `str` 字元，而它內部儲存了核苷酸序列作為位元串。`__init__()` 方法的主要工作是以適當的資料初始位元串構造。`__init__()` 呼叫 `_compress()` 執行將提供的核苷酸 `str` 實際轉換成位元串的苦差事。

請注意，雖然 _compress() 的開頭是底線，但是 Python 實際上並沒有私用方法或變數的概念（所有變數和方法皆可透過反射加以存取，沒有嚴格的隱私強制）。開頭的底線只是約定俗成的用法，表示參與者不應依賴類別外部的方法實作（它可能會改變，應該視為私用）。

> **TIP** 如果你所執行的程式類別裡的方法或實體變數的名稱是以 2 個底線開頭，Python 將會「改變它的名稱」，略施小計來改變它的實作名稱，並且不讓其他類別很容易就發現。我們在這本書使用 1 條底線來表示「私用」的變數或方法，但如果你真的想要強調某些部分的確是私用，或許就希望使用 2 條底線。更多關於 Python 命名的資訊，請查閱 PEP 8 裡的“Descriptive Naming Styles”：http://mng.bz/NA52。

接著，我們來看看實際上要如何執行上述的壓縮。

程式 1.11 trivial_compression.py 承上

```
def _compress(self, gene: str) -> None:
    self.bit_string: int = 1  # 以哨兵標記作為開端
    for nucleotide in gene.upper():
        self.bit_string <<= 2  # 往左位移 2 位元
        if nucleotide == "A":  # 將最後 2 位元改成 00
            self.bit_string |= 0b00
        elif nucleotide == "C":  # 將最後 2 位元改成 01
            self.bit_string |= 0b01
        elif nucleotide == "G":  # 將最後 2 位元改成 10
            self.bit_string |= 0b10
        elif nucleotide == "T":  # 將最後 2 位元改成 11
            self.bit_string |= 0b11
        else:
            raise ValueError("Invalid Nucleotide:{}".format(nucleotide))
```

_compress() 方法相繼查看核苷酸 str 裡的每個字元。當它看到 A，就將 00 加到位元串，當它看到 C，就加入 01，依此類推。請記住，每個核苷酸需要 2 個位元。因此當我們加入每個新的核苷酸之前，要將位元串向左移 2 個位元（self.bit_string <<= 2）。

每個核苷酸都是利用「或」運算（|）加入，而在左移之後，會在位元串的右側加入 2 個 0。若以任何其他值與 0 進行“ORing”（例如 `self.bit_string |= 0b10`）位元運算，會導致另一個值替換掉 0。也就是說，我們不斷加入 2 個新的位元到位元串右側，這 2 個新加位元是由核苷酸的類型所決定。

最後，我們將會實作解壓縮和使用它的特殊 `__str__()` 方法。

程式 1.12 trivial_compression.py 承上

```
def decompress(self) -> str:
    gene: str = ""
    for i in range(0, self.bit_string.bit_length() - 1, 2):
     # - 1 to exclude sentinel
        bits: int = self.bit_string >> i & 0b11
         # get just 2 relevant bits
        if bits == 0b00:  # A
            gene += "A"
        elif bits == 0b01:  # C
            gene += "C"
        elif bits == 0b10:  # G
            gene += "G"
        elif bits == 0b11:  # T
            gene += "T"
        else:
            raise ValueError("Invalid bits:{}".format(bits))
    return gene[::-1]  # [::-1] reverses string by slicing backward
def __str__(self) -> str:  # string representation for pretty printing
    return self.decompress()
```

`decompress()` 每一次會從位元串讀取 2 個位元，並且使用這 2 個位元來決定要將哪個字元加到表示基因的 `str` 的結尾。因為是往後讀取位元，所以相較於壓縮它們的順序（是從右到左而非從左到右），`str` 最終會相反（使用切片記法來反轉 `[::-1]`）。最後請注意便利的 `int` 方法 `bit_length()` 是如何有助於 `decompress()` 的開發。讓我們來試試。

程式 1.13 trivial_compression.py 承上

```
if __name__ == "__main__":
    from sys import getsizeof
    original: str =
     "TAGGGATTAACCGTTATATATATATAGCCATGGATCGATTATATAGGGATTAACCGTTATATATA
     TATAGC CATGGATCGATTATA" * 100
```

```
print("original is {} bytes".format(getsizeof(original)))
compressed: CompressedGene = CompressedGene(original)  # 壓縮
print("compressed is {} bytes".format(getsizeof(
 compressed.bit_string)))
print(compressed)  # 解壓縮
print( "original and decompressed are the same: {}" .format(
 original == compressed.decompress()))
```

使用 sys.getsizeof() 方法，我們可以在輸出表明是否真的透過這項壓縮方案節省了幾乎 75%的基因儲存記憶體成本。

程式 1.14　trivial_compression.py 輸出

```
original is 8649 bytes
compressed is 2320 bytes
TAGGGATTAACC...
original and decompressed are the same: True
```

NOTE　我們在 CompressedGene 類別裡大量使用 if 陳述式在壓縮和解壓縮方法的一系列情況之間做出決定。因為 Python 沒有 switch 陳述式，所以這有點特別。有時你也會在 Python 看到高度依賴字典代替大量的 if 陳述式來處理整組需要判斷的情況。例如，我們可以想像一下查找每個核苷酸各自位元的字典，雖然有時候這可以更具可讀性，但隨之而來的卻可能是效能成本。即使字典查找就技術而言就是 O(1)，但執行雜湊函式的成本有時候意味著字典的效能比一連串的 if 還低。是否真的如此將取決於特定程式的 if 陳述式所需估算的內容來決定。如果你需要在程式碼關鍵部分的 if 和字典查找兩者之間有所抉擇，可能會想對這兩種方法執行效能測試。

1.3　無法破解的加密

單次密碼本（one-time pad，OTP）加密法是一種資料加密的方式，作法是將欲加密的資料與無意義且隨機虛設的資料加以組合，如此一來就一定要同時存取加密結果和虛設的資料，才能重構原本的資料。本質上，這允許加密器具有金鑰對，一把金鑰是加密結果，另一把是隨機虛設的資料。一把金鑰本身並沒有作用，唯有這兩把金鑰的組合才能解開原始資料。如果正確執行，單次密碼本加密法是一種無法破解的加密形式。圖 1.6 展示了整個過程。