導讀

這份導讀讓你可以更了解如何使用本書。

新舊版差異

雖然 Python 是動態定型語言，然而 Python 3.5 正式納入型態提示特性至今的這段日子，已證實對於程式碼的正確性、可讀性等方面有很大的幫助，為了示範如何標註型態，4.3 節會初探型態提示，以及 mypy 之使用，後續範例專案的程式碼都適當地加上了型態，6.4 節會入門泛型，14.5 節談到了進階的泛型觀念，像是型態邊界、共變性、逆變性等。

如果你對型態提示不感興趣，可以忽略本書範例程式碼中的型態提示，也就是刪掉:、->及右方之型態，這不影響程式碼之執行。

本書改版時的第二個重點放在第 13 章，當中增加了非同步之討論，為此，既有的內容與範例都做了調整，並增加 13.3 節，其中包括了 concurrent.futures 模組的介紹，async、await 關鍵字的由來與 asyncio 模組的討論等。

第 11 章的改版重點則有，在 11.3 規則表示式中，增加擴充標記之介紹；新增 11.5，討論 urllib 套件之使用，當中示範了如何下載 HTML 網頁進行分析，作為對 HTML 網頁分析之補充，也增加了附錄 C 談談 Beautiful Soup 之使用。

其他還有一些 Python 新增小特性之介紹與應用，像是格式化字串實字、數字底線等，第 14 章裝飾器增加了@functools.wraps 之應用，相關範例做了些調整等。

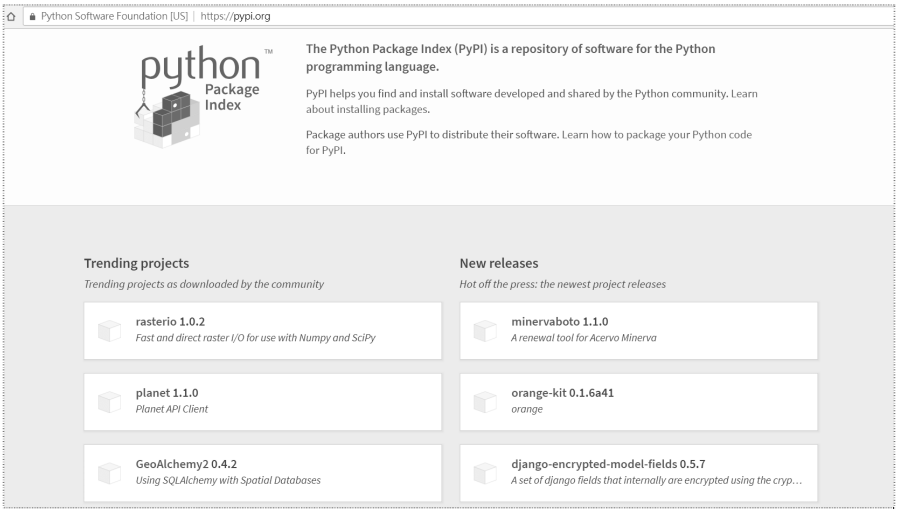

圖 6.4 PyPI 網站

你可以在 PyPI 上尋找適用你需求的套件，想要安裝上面的套件，可以透過 `pip` 來安裝，在 4.3.3 曾經介紹 `pip` 的基本使用方式，可以回顧一下。

6.4 泛型入門

到目前為止，你已經看過不少型態提示的範例了，使用型態來約束程式碼的組織方式是個雙面刃，運用良好可增加可讀性與程式的穩固性，運用失當會使得程式碼充滿對人類無意義的型態資訊，造成程式碼難以閱讀，在運用泛型（Generics）進行型態標註時，更需要在用與不用之間作出衡量。

泛型在運用上有一定的複雜性，對於動態定型的 Python 而言，優勢在於採用鴨子定型而帶來的彈性，多數的情況下也應該這麼做，如果你打算略過這個小節，對撰寫 Python 來說，不會有任何影響。

然而，若開始考慮到 4.3.4 中的一些因素，或者其他工程上的考量，而開始使用型態提示，後續也許就會開始考慮泛型，以便結合一些工具，對程式專案做出更進一步的約束，這就是接下來要討論泛型的目的。

6.4.1　定義泛型函式

先來看一個案例，你的應用程式中使用 list 來收集 int 或者 str，你發現在程式演算過程中，經常要取得 list 中第一個元素，因此定義了 first() 函式，可以傳回 list 的首個元素：

```
def first(lt):
    return lt[0]
```

first([123, 456, 789]) 就會傳回 123，first(['Justin', 'Monica', 'Irene']) 會傳回 'Justin'，問題來了，應用程式是個多人合作的專案，其他開發者會呼叫 first() 函式，其中有開發者忽略了規範，將字串傳入，例如 first('Justin')，試圖取得字串首個字元，因為字串也可以使用索引方式存取內含字元，因此程式也沒有出錯。

然而，規範希望其他開發者呼叫 first() 時只傳入 list，因此你加入了型態提示：

```
def first(lt: list):
    return lt[0]
```

透過 mypy 之類的工具進行型態檢查，傳入字串的開發者收到了型態錯誤的訊息，因此明白 first() 只能傳入 list 了，就這麼相安無事了一陣子，然後有一天發現，有開發者傳入了 ['a', 1, 2,'Justin'] 這樣的引數，也就是 list 中的元素並不是單一型態，而可能有各種型態。

若有夠好的理由，list 有異質型態的元素並非不行，然而，若這並不在團隊規範之內，你的 first() 接受的 list，元素應該是同質型態，要嘛全部是 int，要嘛全部是 str，只不過在型態提示時，使用 List[int]、List[str] 都不對，前者限制 list 的元素只能用 int，後者限制為 str，這時該怎麼辦？

如果有個佔位型態 T 就好了，這樣就可以限制為 List[T]，也就是 list 的元素必須都是 T 型態，對於這類的需求，可以透過 typing 模組的 TypeVar 來定義佔位型態 T：

```
from typing import TypeVar, List

T = TypeVar('T')

def first(l: List[T]) -> T:
    return l[0]
```

這就建立了一個泛型函式，目前的 T 代表著任意型態，如果想限制 T 實際的型態都只能是 int，或者都是 str，可以使用 TypeVar 指定：

```
from typing import TypeVar, List

T = TypeVar('T', int, str)

def first(l: List[T]) -> T:
    return l[0]
```

這麼一來，若試圖使用['a', 1, 2, 'Justin']呼叫 first()，使用 mypy 檢查時就會出現錯誤，只能使用[1, 2, 3]或者['a', 'b', 'c']等，才可以通過型態檢查。

6.4.2 定義泛型類別

假設你實作了 Basket 類別，可以在其中放置物品，例如 Apple 的實例之類的：

```
class Basket:
    def __init__(self):
        self.things = []

    def add(self, thing):
        self.things.append(thing)

    def get(self, idx):
        return self.things[idx]

class Apple:
    pass

basket = Basket()
basket.add(Apple())
apple = basket.get(0)
```

目前的 Basket 實例，可以放置的水果種類是沒有限制的，若想限制只能放置同樣的水果，類似地，若有個 T 佔位型態，可以標註在 add()方法的 things 參數，以及 get()的傳回值型態，就可以達到目的，為了這類需求，typing 模組中提供了 Generics 類別，搭配 TypeVar 就可以定義泛型類別：

```
from typing import TypeVar, Generic

T = TypeVar('T')
```

```
class Basket(Generic[T]):
    def __init__(self):
        self.things = []

    def add(self, thing: T):
        self.things.append(thing)

    def get(self, idx) -> T:
        return self.things[idx]

class Apple:
    pass

basket = Basket[Apple]()
basket.add(Apple())
apple: Apple = basket.get(0)
```

在定義泛型類別時，除了使用 TypeVar 定義型態 T 之外，可以令類別繼承 Generic[T]，在類別之中就可以使用 T 來定義參數、傳回值等之型態；在實例化 Basket 時，可以指定 T 的型態，例如 Basket[Apple]()，這麼一來，後續在呼叫相關方法時，就可以透過 mypy 來檢查方法可以接受的參數或傳回值型態。

提示 »» 技術上而言，Generic 使用了一個定義了__getitem__()方法的 meta 類別，透過繼承，它的子類別就可以使用[]運算取值，有關於__getitem__()，第 9 章會討論，第 12 章會討論 meta 類別。

以上面的範例來說，若程式中定義了 Banana 類別，而程式碼中撰寫了 basket.add(Banana())的話，在使用 mypy 進行型態檢查時，就會出現錯誤訊息。

在定義泛型類別時，可以使用的型態佔位名稱，並不限於一個，例如在 typing 模組的文件說明中，就有著這個範例：

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

chapter

6 類別的繼承

學習目標

- 瞭解繼承目的
- 認識鴨子定型
- 重新定義方法
- 認識 `object`
- 建立、尋找文件資源
- 泛型入門

6.1 何謂繼承？

物件導向中，子類別繼承（Inherit）父類別，避免重複的行為與實作定義，不過並非為了避免重複定義行為與實作就得使用繼承，濫用繼承而導致程式維護上的問題時有所聞，如何正確判斷使用繼承的時機，以及繼承之後如何活用多型，才是學習繼承時的重點。

6.1.1 繼承共同行為

繼承基本上是為了避免多個類別間重複實作相同的行為。以實際的例子來說明比較清楚，假設你在正開發一款 RPG（Role-playing game）遊戲，一開始設定的角色有劍士與魔法師。首先你定義了劍士類別：

```
class SwordsMan:
    def __init__(self, name: str, level: int, blood: int) -> None:
        self.name = name   # 角色名稱
        self.level = level # 角色等級
        self.blood = blood # 角色血量

    def fight(self):
        print('揮劍攻擊')

    def __str__(self):
        return "('{name}', {level}, {blood})".format(**vars(self))

    def __repr__(self):
        return self.__str__()
```

劍士擁有名稱、等級與血量等屬性，可以揮劍攻擊，為了方便顯示劍士的屬性，定義了__str__()方法，並讓__repr__()的字串描述直接傳回了__str__()的結果。

__str__()方法中直接使用了 5.2.3 介紹過的 vars()函式，以 dict 取得目前實例的屬性名稱與值，然後用了 4.2.2 介紹過的 dict 拆解方式，將 dict 的屬性名稱與值拆解後，傳給字串的 format()方法，這樣的寫法相對於以下，顯然簡潔了許多：

```
class SwordsMan:
    略…
    def __str__(self):
        return "('{name}', {level}, {blood})".format(
```

```
            name = self.name, level = self.level, blood = self.blood)
```

略…

接著你為魔法師定義類別：

```
class Magician:
    def __init__(self, name: str, level: int, blood: int) -> None:
        self.name = name   # 角色名稱
        self.level = level # 角色等級
        self.blood = blood # 角色血量

    def fight(self):
        print('魔法攻擊')

    def cure(self):
        print('魔法治療')

    def __str__(self):
        return "('{name}', {level}, {blood})".format(**vars(self))

    def __repr__(self):
        return self.__str__()
```

你注意到什麼呢？因為只要是遊戲中的角色，都會具有角色名稱、等級與血量，也定義了相同的__str__()與__repr__()方法，Magician 中粗體字部份與 SwordsMan 中相對應的程式碼重複了。

重複在程式設計上，就是不好的訊號。舉個例子來說，如果要將 name、level、blood 更改為其他名稱，那就要修改 SwordsMan 與 Magician 兩個類別，如果有更多類別具有重複的程式碼，那就要修改更多類別，造成維護上的不便。

如果要改進，可以把相同的程式碼提昇（Pull up）至父類別 Role，並讓 SwordsMan 與 Magician 類別都繼承自 Role 類別：

game1_rpg.py

```
class Role:  ◀── ❶ 定義類別 Role
    def __init__(self, name: str, level: int, blood: int) -> None:
        self.name = name   # 角色名稱
        self.level = level # 角色等級
        self.blood = blood # 角色血量

    def __str__(self):
        return "('{name}', {level}, {blood})".format(**vars(self))

    def __repr__(self):
```

實際上，在 4.3.2 談到可以使用 List[str]來標註型態之時，就已經實際應用了 typing 模組中，預先定義好的泛型類別了；對於支援泛型的類別，如果在使用時不指定佔位型態 T 實際之型態，會使用 typing 模組的 Any 型態，而不是 object，前者可以支援鴨子定型，後者就真的限定為 object。

直接使用範例來比較 Any 與 object 的不同，首先是被標註為 List[object]的情況：

```
from typing import List
lt: List[object] = ['1', '2', '3']
print(lt[0].upper())
```

若使用 mypy 檢查型態的話，上面的範例第三行會出現 error: "object" has no attribute "upper"的訊息，因為限定為 object，而 object 本身並沒有定義 upper()方法；若是底下就不會出錯：

```
from typing import Any, List
lt: List[Any] = ['1', '2', '3']
print(lt[0].upper())
```

List[Any]實際上也可以寫為 List，結果就也等同於 list，因此上面的範例中，標註為 lt: list 也是可以的。

就身為動態定型的 Python 而言，以上的討論加上 typing 模組，應該足以應付大多數想要自定義泛型的需求，然而泛型實際上還有不少可以深入的地方，這在第 14 章會再加以討論。

6.5 重點複習

類別名稱旁邊多了個括號，並指定了類別，這在 Python 中代表著繼承該類別。

鴨子定型實際的意義在於：「思考物件的行為，而不是物件的種類。」依照此思維設計的程式，會具有比較高的通用性。

在繼承後若打算基於父類別的方法實作來重新定義某個方法，可以使用 super()來呼叫父類別方法。

如果希望子類別在繼承之後，一定要實作的方法，可以在父類別中指定 metaclass 為 abc 模組的 ABCMeta 類別，並在指定的方法上標註 abc 模組的

@abstractmethod 來達到需求。抽象類別不能用來實例化，繼承了抽象類別而沒有實作抽象方法的類別，也不能用來實例化。

若沒有指定父類別，那麼就是繼承 object 類別。

在 Python 中若沒有定義的方法，某些場合下必須呼叫時，就會看看父類別中是否有定義，如果定義了自己的方法，那麼就會以你定義的為主，不會主動呼叫父類別的方法。

在 Python 3 中，在定義方法時使用無引數的 super()呼叫，等同於 super(__class__, <first argument>)呼叫，__class__代表著目前所在類別，而<first argument>是指目前所在方法的第一個引數。

就綁定方法來說，在定義方法時使用無引數的 super()呼叫，而方法的第一個參數名稱為 self，就相當於 super(__class__, self)。

在 object 類別上定義了__lt__()、__le__()、__eq__()、__ne__()、__gt__()、__ge__()等方法，這組方法定義了物件之間使用<、<=、==、!=、>、>=等比較時，應該要有的比較結果，這組方法在 Python 官方文件上，被稱為 Rich comparison 方法。

想要能使用==來比較兩個物件是否相等，必須定義__eq__()方法，因為__ne__()預設會呼叫__eq__()並反相其結果，因此定義了__eq__()就等於定義了__ne__()，也就可以使用!=比較兩個物件是否不相等。

object 定義的__eq__()方法，預設是使用 is 來比較兩個物件，實作__eq__()時通常也會實作__hash__()。

__lt__()與__gt__()互補，而__le__()與__ge__()互補，因此基本上只要定義__gt__()、__ge__()就可以了。

從 Python 3.4 開始新增了 enum 模組。

在 Python 中可以進行多重繼承，也就是一次繼承兩個父類別的程式碼定義，父類別之間使用逗號作為區隔。

一個子類別在尋找指定的屬性或方法名稱時，會依據類別的__mro__屬性的 tuple 中元素順序尋找，如果想知道直接父類別的話，則可以透過類別的__bases__來得知。

判定一個抽象方法是否有實作，也是依照__mro__中類別的順序。

多重繼承的能力，通常建議只用來繼承 ABC，也就是抽象基礎類別，一個抽象基礎類別，不會定義屬性，也不會有__init__()定義。

在函式、類別或模組定義的一開頭，使用'''包括起來的多行字串，會成為函式、類別或模組的__doc__屬性值，也就是會成為 help()的輸出內容之一。

在執行 python 時指定-m 引數，表示執行指定模組中頂層的程式流程。

Python 官方維護的網站，可以做為搜尋程式庫時不錯的起點。

6.6 課後練習

實作題

1. 雖然目前不知道要採用的執行環境會是文字模式、圖型介面或者是 Web 頁面，然而，現在就要你寫出一個猜數字遊戲，會隨機產生 0 到 9 的數字，程式可取得使用者輸入的數字，並與隨機產生的數字相比，如果相同就顯示「猜中了」，如果不同就繼續讓使用者輸入數字，直到猜中為止。請問你該怎麼做？

2. 在 5.3.4 曾經開發過一個 Rational 類別，請為該類別加上 Rich comparison 方法的實作，讓它可以有 >、>=、<、<=、==、!= 的比較能力。