## Effective Python 的各界好評

「Slatkin 著的 *Effective Python* 中每一個主題都是自成一體的一堂課，附有自己的程式原始碼。這讓你能夠隨意翻閱此書：其中的主題易於瀏覽，並可以讀者需要的順序來研讀。我會向 Python 的學員推薦 *Effective Python* 這本書，它以相當精簡的方式包含了非常廣泛的主題，為中階的 Python 程式設計師提供了主流的建議做法。」

—*Brandon Rhodes*，*Dropbox* 軟體工程師及 *PyCon* 2016-2017 主席

「我在 Python 程式設計上有多年的經驗，我以為已經很了解它了。我得感謝這本訣竅和技巧的寶庫，讓我知道還有許多方式可以改善我的 Python 程式碼，讓它跑得更快（例如使用內建資料結構）、更容易讀（例如使用強制性的 keyword-only 引數），並且讓它們更為 Pythonic（例如使用 zip 來平行迭代串列）。」

—*Pamela Fox*，*Khan Academy* 教育工程師

「如果當初從 Java 轉換到 Python 時有這本書可讀，就能節省我重複修改程式碼那好幾個月的時間了，每次改寫都是因為發現了我以『非 Pythonic』的方式在編寫程式。這本書將許多 Python 基本的『must-knows』主題都收集在一起，讓你不必花費好幾個月甚或數年的時間一一遭遇那些問題才去尋求解法。這本書所涵蓋的範圍之廣令人印象深刻，從 PEP8 的重要性到 Python 的慣用法，還談到函式、方法以及類別的設計、如何有效利用標準程式庫、高品質的 API 設計、測試與效能評估，這本書裡通通都有。它能讓 Python 初學者或有經驗的開發者知道怎樣才稱得上真正的 Python 程式設計師。」

—*Mike Bayer*，*SQLAlchemy* 創立者

「*Effective Python* 會讓你的 Python 技能提升到更高的層次，它清楚明瞭的指引，能夠幫你改善 Python 程式碼的編碼風格與功能。」

—*Leah Culver*，*Dropbox Developer Advocate*

「如果你是其他程式語言的資深開發者，想要快速地抓住 Python 的精要，掌握基本的語言構造，寫出更為 Pythonic 的程式碼，那麼這本書就是非常好的參考資源。此書的組織明確精簡、容易消化，而其中的每個主題與章節都自成一體，都是針對特定面向深思熟慮之後的精華。此書廣泛地涵蓋純 Python 的語言構造，不會讓龐大又複雜的 Python 生態系統把讀者搞得頭昏腦脹。對於資深開發人員，這本書也會以範例深入探討他們之前可能沒碰過的語言構造，並提供例子來說明較不常用的語言特色。很明顯地，作者本人對 Python 非常熟練，他用他的專業經驗來提示讀者常會碰到但難以察覺的臭蟲以及常見的失敗模式。此外，這本書另外一個優越之處在於它明確指出了 Python 2.X 與 Python 3.X 之間的細微差異，你要在不同 Python 版本之間轉換時，可以拿它來好好複習一番。」

—*Katherine Scott*，*Tempo Automation* 軟體部門主管

「對於初學者或有經驗的程式設計師而言，這都是一本非常好的參考書。程式碼範例與說明都是經過仔細思量之後的成果，解說精簡且完整。」

—*C. Titus Brown*，*UC Davis* 副教授

「這是一本相當實用的參考資源，教你進階的 Python 用法，以及如何建置簡潔、更容易維護的軟體。想要提升他們 Python 程式技能的人只要將此書的建議納入實務，就能夠從中獲益。」

—*Wes McKinney*，*pandas* 的創立者、*Python for Data Analysis* 的作者以及 *Cloudera* 的軟體工程師

# 類別與繼承

# 3

作為一個物件導向（object-oriented）程式語言，Python 支援完整的物件功能，例如繼承（inheritance）、多型（polymorphism），以及封裝（encapsulation）。要在 Python 中完成某些事情，通常需要撰寫新的類別（classes），並且定義它們如何透過它們的介面（interfaces）與階層架構（hierarchies）來互動。

Python 的類別與繼承讓你能輕易地以物件來表達程式的預期行為。它們能讓你隨著時間的演進改善並擴充其功能性。在需求不斷改變的環境中，提供了應變的彈性。知道如何使用它們，能讓你寫出容易維護的程式碼。

## 做法 22 優先選用輔助類別而非使用字典或元組來管理記錄

Python 的內建字典型別（dictionary type）非常適合用來維護一個物件生命期（lifetime）中的動態內部狀態。這裡的**動態**（*dynamic*）的意思是，你得記錄無法預期的一組識別符（identifiers）的那種情況。舉例來說，假設你想要記錄一些學生的成績，但事先並不知道他們的姓名。你可以定義一個類別來將那些名字儲存在一個字典中，而非為每個學生使用一個預先定義的屬性（predefined attribute）。

```
class SimpleGradebook(object):
    def __init__(self):
        self._grades = {}

def add_student(self, name):
        self._grades[name] = []
```

```
def report_grade(self, name, score):
    self._grades[name].append(score)

def average_grade(self, name):
    grades = self._grades[name]
    return sum(grades) / len(grades)
```

這個類別的使用方式很簡單。

```
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
# ...
print(book.average_grade('Isaac Newton'))
>>>
90.0
```

字典太容易使用，以至於有著過度擴充它們而寫出脆弱程式碼的危險存在。舉例來說，假設你想要擴充 SimpleGradebook 類別，讓它維護一個串列來記錄以科目區分的成績，而非只有整體成績。你可以修改 _grades 字典來將學生姓名（也就是鍵值）映射（map）到另一個字典（存放那些值）。最內層的字典會將科目（鍵值）映射到成績（值）。

```
class BySubjectGradebook(object):
    def __init__(self):
        self._grades = {}
    def add_student(self, name):
        self._grades[name] = {}
```

這看起來似乎很簡單。report_grade 與 average_grade 方法會因為要處理多層字典而增加一點複雜度，但仍然在可以處理的範圍內。

```
    def report_grade(self, name, subject, grade):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append(grade)

    def average_grade(self, name):
        by_subject = self._grades[name]
        total, count = 0, 0
        for grades in by_subject.values():
```

```
            total += sum(grades)
            count += len(grades)
        return total / count
```

這個類別的使用方式依然很簡單。

```
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
```

現在，想像一下如果你的需求再次改變了。也想要記錄每個分數相對於班級整體成績的權重（weight），讓期中考與期末考的重要性比隨堂測驗還要高。實作這種功能的方式之一是變更最內層的字典，不是把科目（鍵值）映射到成績（值），我可以用 `(score, weight)` 這個元組（tuple）來作為值。

```
class WeightedGradebook(object):
    # ...
    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject.setdefault(subject, [])
        grade_list.append((score, weight))
```

雖然對於 `report_grade` 的修改看似簡單，只是讓值變為一個元組而已，但現在 `average_grade` 方法得在一個迴圈中再使用另一個迴圈，變得難以閱讀。

```
    def average_grade(self, name):
        by_subject = self._grades[name]
        score_sum, score_count = 0, 0
        for subject, scores in by_subject.items():
            subject_avg, total_weight = 0, 0
            for score, weight in scores:
                # ...
        return score_sum / score_count
```

這個類別的使用方式也變得更困難了。那些位置引數（positional arguments）上的數字所代表的意義，並不容易弄清楚。

```
book.report_grade('Albert Einstein', 'Math', 80, 0.10)
```

當你看到像這樣的複雜情況出現時，就知道該是時候放棄字典與元組，改為使用形成階層架構的類別（a hierarchy of classes）了。

一開始，你並不知道你得支援有權重的成績，所以新增額外輔助類別（helper classes）的功夫似乎不值得。Python 內建的字典與元組型別就能輕易地應付這種情況，只要加入一層又一層的內部記錄就行了。但你應該避免內嵌超過一層（也就是避免含有字典的字典）。那會使得其他程式設計師難以閱讀你的程式碼，並造成維護上的夢魘。

當你發現記錄的工作變得複雜，就將它拆解成多個類別。這讓你得以提供定義良好的介面，以較好的方式來封裝你的資料。這也能讓你在介面與具體實作（concrete implementations）之間建立一個抽象層。

## 重構為類別

你可以從依存樹（dependency tree）的底部開始改寫為類別，也就是單一的成績。對於這種簡單的資訊來說，使用一個類別似乎太過，而一個元組（tuple）看起來好像比較適當，因為成績是不可變的（immutable）。這裡我使用 `(score, weight)` 這種元組在一個串列中記錄成績：

```
grades = []
grades.append((95, 0.45))
# ...
total = sum(score * weight for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

問題在於一般的元組是位置型（positional）。當你想要將更多的資訊關聯到一個成績，像是老師的評語之類的，就得重新改寫用到那個二元組（two-tuple）的地方，讓程式碼知道現在有三個項目存在，而非只是兩個。在此，我使用 _（底線變數名稱，Python 對於沒用到的變數的慣用語法）來捕捉元組中的第三個項目，並直接忽略它：

```
grades = []
grades.append((95, 0.45, 'Great job'))
# ...
total = sum(score * weight for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

這種將元組擴充得越來越長的模式，就類似越來越多層的字典。只要發現所用元組的長度超過了二元組，就該考慮其他的做法了。

collections 模組中的 namedtuple 型別能做的事情正是你需要的。它能讓你輕易地定義小型、不可變的資料類別。

```
import collections
Grade = collections.namedtuple('Grade', ('score', 'weight'))
```

這些類別能以位置引數或關鍵字引數的方式來建構。那些欄位能以具名屬性（named attributes）來存取。擁有具名屬性可以讓你在之後有變更的需求，且得新增行為到簡單的資料容器之時，從一個 namedtuple 輕易移至你自己的類別。

**namedtuple 的限制**

雖然在許多情況中都有用處，但理解何時 namedtuple 帶來的壞處多過於好處，仍然很重要。

- 你無法為 namedtuple 類別指定預設的引數值。如果你的資料有許多選擇性的特性（optional properties），那麼這就會使得它們變得過於笨重。如果發現你所使用的屬性不只少數幾個，那麼定義你自己的類別或許會是比較好的選擇。
- namedtuple 實體（instances）的屬性值仍然可透過數值的索引（numerical indexes）與迭代（iteration）來存取。特別是在對外提供的 API 中，這可能會導致非預期的使用，讓你在之後更難以改寫為一個真正的類別。如果你無法控制你的 namedtuple 實體會被如何使用，最好還是定義你自己的類別。

接著，你可以撰寫一個類別來表示含有一組成績的單一科目。

```
class Subject(object):
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
```

```
        self._grades.append(Grade(score, weight))
    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

然後你會撰寫一個類別來表示單一位學生選讀的科目所組成的一個集合。

```
class Student(object):
    def __init__(self):
        self._subjects = {}

    def subject(self, name):
        if name not in self._subjects:
            self._subjects[name] = Subject()
        return self._subjects[name]

    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```

最後，你會為所有的學生撰寫一個容器（container），並以他們的姓名作為鍵值。

```
class Gradebook(object):
    def __init__(self):
        self._students = {}

    def student(self, name):
        if name not in self._students:
            self._students[name] = Student()
    return self._students[name]
```

這些類別的行數幾乎是之前實作的兩倍，但這些程式碼比較容易讀。這些類別的使用範例同樣也比較清楚且容易擴充。

```
book = Gradebook()
albert = book.student('Albert Einstein')
math = albert.subject('Math')
math.report_grade(80, 0.10)
# ...
print(albert.average_grade())

>>>
81.5
```

如果必要，你可以撰寫回溯相容（backwards-compatible）的方法，來協助將舊 API 的使用方式移植到物件階層架構的新方式。

**請記住**

- 避免製作其中的值為其他字典或是過長元組的字典。
- 如果尚不需要使用到較有彈性的完整類別，請使用 `namedtuple` 來製作輕量化、不可變的資料容器。
- 如果記錄內部狀態的字典變得太過複雜，就將記錄用的程式碼改寫為使用多個輔助類別。

## 做法 23 接受函式作為簡單的介面，而非使用類別

許多 Python 的內建 API 都允許你傳入一個函式來自訂行為。這些**掛接器**（*hooks*）會在 API 執行的時候，被用來回呼（call back）你的程式碼。舉例來說，`list` 型別的 `sort` 方法接受一個選擇性的 `key` 引數，它會被用來決定每個索引用於排序的值。在此，我提供了一個 `lambda` 運算式作為 `key` 這個掛接器，以依據名稱長度來排序一個由名稱所組成的串列：

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=lambda x: len(x))
print(names)

>>>
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
```

在其他的語言中，你可能會預期這種掛接器會由一個抽象類別（abstract class）來定義。在 Python 中，有許多的掛接器都只是帶有定義明確的引數及回傳值的無狀態函式（stateless functions）。函式很適合用來當作掛接器，因為比起類別，它們更容易描述，定義也比較簡單。函式能當作掛接器來用，是因為 Python 具有一級（*first-class*）函式：在 Python 中，函式與方法可被傳來傳去，並且可像其他值那樣被參考。

舉例來說，假設你想要自訂 `defaultdict` 類別（參見做法 46：「使用內建的演算法與資料結構」）的行為。這個資料結構允許你提供一個函式，每次只要存取到缺少的鍵值，這個函式就會被呼叫。這個函式必須回傳字典中那個缺少的鍵值應該有的預設值。在此，我定義了一個掛接器，它會在每次鍵值有缺時，進行記錄，並回傳 `0` 作為預設值：

```
def log_missing():
    print('Key added')
    return 0
```

只要給定一個初始字典，以及一組期望的增量（increments），我就能使 `log_missing` 函式執行並印出兩次訊息（一次為 `'red'`，另一次為 `'orange'`）。

```
current = {'green': 12, 'blue': 3}
increments = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9),
]

result = defaultdict(log_missing, current)
print('Before:', dict(result))
for key, amount in increments:
    result[key] += amount
print('After: ', dict(result))

>>>
Before: {'green': 12, 'blue': 3}
Key added
Key added
After: {'orange': 9, 'green': 12, 'blue': 20, 'red': 5}
```

提供像是 log_missing 這樣的函式可讓 API 容易建置和測試，因為這將副作用（side effects）與確定性的行為（deterministic behavior）區隔開來了。舉例來說，假設你現在希望這個傳入 defaultdict 的預設值掛接器計算缺少的鍵值總數。達成這個目的的一個方式是使用一個有狀態的閉包（stateful closure，參見做法 15：「知道 Closures 如何與變數範疇互動」）。這裡，我定義了一個輔助函式，它就使用這樣的一個 closure 作為預設值掛接器：

```
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count # 有狀態的閉包 (stateful closure)
        added_count += 1
        return 0

    result = defaultdict(missing, current)
    for key, amount in increments:
        result[key] += amount

    return result, added_count
```

執行這段程式碼會產生預期的結果（2），雖然 defaultdict 絲毫不知 missing 掛接器有保存狀態。這就是接受簡單函式作為介面的另一個好處。藉由把狀態隱藏在一個 closure 中，讓我們之後能夠輕易地加入新功能。

```
result, count = increment_with_report(current, increments)
assert count == 2
```

定義一個 closure 作為有狀態的掛接器（stateful hooks）的問題在於，它比無狀態函式的範例還要來得難讀。另外的做法是定義一個小型類別來封裝你想要記錄的狀態。

```
class CountMissing(object):
    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0
```

在其他語言中，你可能會想說現在我們必須修改 defaultdict 才能配合 CountMissing 的介面。但在 Python 中，由於函式是第一級的（first-class），你能夠直接在一個物件上參考 CountMissing.missing 方法，並將它傳入 defaultdict 作為預設值掛接器（default value hook）。要把一個方法用在一個函式介面上，是很容易的事情。

```
counter = CountMissing()
result = defaultdict(counter.missing, current) # 方法參考

for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

使用像這樣的一個輔助類別（helper class）來提供有狀態的 closure 的行為，會比上面的 increment_with_report 函式還要清楚易懂。然而，如果單獨來看，我們仍然無法立即看出 CountMissing 類別的用途為何。誰來建構一個 CountMissing 物件？誰負責呼叫 missing 方法？這個類別需要在未來加入其他的公開方法（public methods）嗎？在你看到它與 defaultdict 搭配使用的情形之前，這個類別可以說是個謎團。

為了把這種情況弄得更清楚一點，Python 允許類別定義 __call__ 這個特殊方法。__call__ 能讓一個物件像函式那樣被呼叫。它也會使 callable 內建函式為這樣的一個實體（instance）回傳 True。

```
class BetterCountMissing(object):
    def __init__(self):
        self.added = 0

    def __call__(self):
        self.added += 1
        return 0

counter = BetterCountMissing()
counter()
assert callable(counter)
```

這裡，我用了一個 BetterCountMissing 實體來作為一個 defaultdict 的預設值掛接器，以記錄我們缺少的鍵值的數目：

```
counter = BetterCountMissing()
result = defaultdict(counter, current) # 仰賴 __call__
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

這比 `CountMissing.missing` 的範例還要清楚得多。`__call__` 代表一個類別的實體會被用在原本適用一個函式引數的地方（例如 API 掛接器）。它導引程式碼的新讀者來到負責該類別主要行為的進入點。它提供了強烈的暗示，指出該類別的目標就是要作為一個有狀態的閉包（a stateful closure）。

最棒的地方是，`defaultdict` 對你使用 `__call__` 時到底發生了什麼事情仍然渾然不知。`defaultdict` 所需要的，就是一個用作預設值掛接器（default value hook）的函式。Python 提供了許多方式讓你可以依據你想達成的事情來滿足一個簡單的函式介面。

**請記住**

- 在 Python 中，除了定義並實體化（instantiating）類別之外，函式也可以用來當作不同元件之間的簡單介面。
- 在 Python 中，對函式與方法的參考（references）都是第一級的（first class），這代表它們能被用在運算式中，就像其他的型別一樣。
- `__call__` 特殊方法能夠讓一個類別的實體被呼叫，就像其他的 Python 函式那樣。
- 當你需要一個函式來保存狀態，請考慮定義一個提供了 `__call__` 方法的類別，而非定義一個有狀態的 closure（參見做法 15：「知道 Closures 如何與變數範疇互動」）。

## 做法 24 使用 @classmethod 多型機制來建構泛用物件

在 Python 中，不只物件支援多型（polymorphism），類別也支援。這表示什麼意義，又有什麼好處呢？

多型這種機制，可以讓同一階層架構（hierarchy）中的多個類別各自實作某個方法的專屬版本。這能夠讓多個類別滿足相同的介面或是抽象基礎類別

（abstract base class），但同時又能提供不同的功能性（範例請見做法 28：「繼承 collections.abc 以建立自訂的容器型別」）。

舉例來說，假設你正在撰寫一個 MapReduce 實作，而你想要用一個共通的類別來表示輸入資料（input data）。這裡，我定義了這樣的一個類別，它帶有一個必須由其子類別（subclasses）來定義的 read 方法：

```
class InputData(object):
    def read(self):
        raise NotImplementedError
```

這裡有 InputData 的一個具體子類別（concrete subclass），它會從磁碟上的一個檔案讀取資料：

```
class PathInputData(InputData):
    def __init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        return open(self.path).read()
```

像 PathInputData 這樣的 InputData 子類別你可以擁有任意的數目，而它們每個都能實作標準的介面，讓 read 回傳要處理的位元組資料。其他的 InputData 子類別可能讀取自網路，或是即時解壓縮資料等。

你可能還會需要為 MapReduce 工作者（worker）建立一個類似的抽象介面（abstract interface），以一種標準方式來消耗輸入資料。

```
class Worker(object):
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

在此，我為 Worker 定義了一個具體子類別來實作我想要套用的特定 MapReduce 功能，也就是一個簡單的 newline 計數器：

```
class LineCountWorker(Worker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')

    def reduce(self, other):
        self.result += other.result
```

這個實作看起來好像很不錯，但我已經面臨了所有這些中的最大關卡。我要怎樣把所有的這些東西拼起來呢？我有一組很不錯的類別，它們帶有合理的介面和抽象層，不過那些只有在物件被建構起來之後才能發揮用處。什麼要負責建構這些物件，並指揮 MapReduce 的運行？

最簡單的方式是以一些輔助函式（helper functions）手動建置並連接這些物件。這裡我列出了一個字典的內容，並為其中所包含的每個檔案建構了一個 PathInputData 實體：

```
def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))
```

接著，我使用 generate_inputs 所回傳的 InputData 實體建立了 LineCountWorker 實體。

```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

我把 map 步驟分散到多個執行緒（threads，參見做法 37：「執行緒用於阻斷式的 I/O，避免用於平行處理」）上來執行這些 Worker 實體。然後，我重複地呼叫 reduce 來將這些結果結合為一個最後的值。

```
def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
```

```
    for thread in threads: thread.join()

    first, rest = workers[0], workers[1:]
    for worker in rest:
        first.reduce(worker)
    return first.result
```

最後，我在一個函式中將所有的這些片段連接在一起，以執行每個步驟。

```
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

在一組測試檔案上執行這個函式，運作起來很順利。

```
from tempfile import TemporaryDirectory

def write_test_files(tmpdir):
    # ...

with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    result = mapreduce(tmpdir)

print('There are', result, 'lines')

>>>
There are 4360 lines
```

那麼問題在哪呢？最大的問題在於，這個 `mapreduce` 函式並非泛用的（generic）。如果你想要撰寫另外的 `InputData` 或 `Worker` 子類別，你就得重寫 `generate_inputs`、`create_workers` 與 `mapreduce` 函式來搭配。

這個問題最根本的癥結是，我們沒有泛用的方式來建構物件。在其他語言中，你可能會以建構器的多型（constructor polymorphism）來解決這個問題，要求 `InputData` 的每個子類別提供一個特殊的建構器，這個建構器可被負責協調 MapReduce 的輔助方法以泛用的方式使用。麻煩的地方在於，Python 只允許單一建構器方法 `__init__`。所以要求每個 `InputData` 子類別都要有一個相容的建構器，並不合理。

解決這個問題的最佳方式是使用 `@classmethod` 的多型。這完全就像我用於 `InputData.read` 的那種實體方法多型（instance method polymorphism），只不過這次是套用到整個類別，而非它們所建構的物件。

讓我將這個想法套用到 MapReduce 類別身上。在此，我以一個泛用的類別方法（generic class method）來擴充 `InputData` 類別，它負責使用一個共通的介面來創建新的 `InputData` 實體：

```
class GenericInputData(object):
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

我讓 `generate_inputs` 接受一個字典，其中含有一組組態參數（configuration parameters），這些參數如何解讀，則交由 `InputData` 的具體子類別去決定。在此，我使用了 `config` 來找出要列出輸入檔案的目錄：

```
class PathInputData(GenericInputData):
    # ...
    def read(self):
        return open(self.path).read()

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

同樣地，我能讓 `create_workers` 輔助器（helper）成為 `GenericWorker` 類別的一部分。在此，我使用 `input_class` 參數，它必須是 `GenericInputData` 的一個子類別，以產生必要的輸入。我把 `cls()` 當作一個泛用建構器（generic constructor）來建構 `GenericWorker` 具體子類別的實體。

```
class GenericWorker(object):
    # ...
    def map(self):
        raise NotImplementedError
```

```
    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers
```

注意到上面對 `input_class.generate_inputs` 的呼叫就是我試著展示的類別多型（class polymorphism）。你也能看到 `create_workers` 呼叫 `cls` 的方式如何在直接使用 `__init__` 方法之外，提供了建構 `GenericWorker` 物件的一個替代方式。

這對我 `GenericWorker` 具體子類別的影響只不過是修改其父類別（parent class）而已。

```
class LineCountWorker(GenericWorker):
    # ...
```

最後，我可以重新改寫 `mapreduce` 函式讓它變成完全泛用的。

```
def mapreduce(worker_class, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)
```

在一組測試檔案上執行這個新的 worker 會產生跟舊有實作同樣的結果。差異在於，這個 `mapreduce` 函式需要更多的參數來讓它可以泛用的方式運作。

```
with TemporaryDirectory() as tmpdir:
    write_test_files(tmpdir)
    config = {'data_dir': tmpdir}
    result = mapreduce(LineCountWorker, PathInputData, config)
```

現在你能以你想要的方式撰寫其他的 `GenericInputData` 與 `GenericWorker` 類別，而不用重新改寫任何接合用的程式碼（glue code）了。