BONUS

A

組合多個 DataFrame 或 Series

Pandas 提供了不同選項來組合多個 DataFrame 或 Series。現在來快速瀏覽一下：

- append()：最不靈活，只允許將新的列添加到 DataFrame。
- concat()：用途最廣，可以沿水平或垂直方向組合任意數量的 DataFrame 或 Series。
- join()：只能沿水平方向組合，會先將原 DataFrame 的索引（或欄位）與另一個 DataFrame 的索引對齊後再組合。
- merge()：只能沿水平方向組合，提供了類似 SQL 的 JOIN 指令功能。

接下來，我們將一一說明以上的不同做法。

A.1 在 DataFrame 上添加新的列

在分析資料時，建立新欄位比建立新的列更為常見。這是因為新的一列資料基本上就是一個新的觀察結果，而作為一名分析師，持續收集新資料並不是你的工作。這個工作通常會在其他平台（例如：關聯式資料庫管理系統）進行。雖然如此，你還是有可能會用到該功能，因此了解如何在現有的 DataFrame 中建立新的列仍是必要的。

在接下來的範例中，我們會先使用 **loc 屬性**在 DataFrame 中添加一列資料，然後再來說明 append() 的做法。

動手做

01 讀入 names 資料集並輸出其內容：

In

```
import pandas as pd
import numpy as np
names = pd.read_csv('data/names.csv')
names
```

Out

```
    Name         Age
0   Cornelia     70
1   Abbas        69
2   Penelope     4
3   Niko         2
```

02 第 6 章曾提過，loc 屬性可根據**索引標籤**與**欄位名稱**來選擇或指派資料。此處我們建立一個包含新資料的**串列**（list），並將索引標籤設為 4。由於原資料中不存在索引標籤為『4』的資料，因此會在 names 中建立一個新的列：

In

```
new_data_list = ['Aria', 1]
names.loc[4] = new_data_list
names
```

Out

```
    Name         Age
0   Cornelia     70
1   Abbas        69
2   Penelope     4
3   Niko         2
4   Aria         1
```

多了一個索引標籤為『4』的列，其內容來自剛剛創建的新資料

⚠ loc 屬性會**就地（in-place）修改** DataFrame 的內容：若原始 DataFrame 存在索引標籤為 4 的列資料，則 new_data_list 中的資料會覆蓋其內容。

03 在步驟 2 中，我們是透過整數索引標籤來添加新的列，但你也可以使用非整數標籤來添加更多的列。無論新標籤的值為何，Pandas 都會將新的列資料添加至 DataFrame 的末端：

In

```
names.loc['five'] = ['Zach', 3] ◄── 新增一列索引標籤為『five』字串的資料
names
```

Out

	Name	Age
0	Cornelia	70
1	Abbas	69
2	Penelope	4
3	Niko	2
4	Aria	1
five	Zach	3

04 雖然我們可以使用串列來添加新資料，但使用字典可以更明確地把『欄位名稱』與『欄位值』關聯在一起。此外，我們可以動態地利用 DataFrame 長度做為新資料的索引標籤：

In

```
names.loc[len(names)] = {'Name':'Zayd', 'Age':2}
names
```

動態指定索引標籤，由於先前的 names 中有 6 筆資料，故新資料的索引標籤為 6

Out

	Name	Age
0	Cornelia	70
1	Abbas	69
2	Penelope	4
3	Niko	2
4	Aria	1
five	Zach	3
6	Zayd	2 ◄── 新增的資料

05 我們也可用 Series 來新增列，該做法與使用字典的效果是相同的：

In

```
names.loc[len(names)] = pd.Series({'Age':32, 'Name':'Dean'})

names
```

欄位名稱的順序可以和 DataFrame 的不同

Out

	Name	Age
0	Cornelia	70
1	Abbas	69
2	Penelope	4
3	Niko	2
4	Aria	1
five	Zach	3
6	Zayd	2
7	Dean	32

06 前面所用的 loc 屬性會對 DataFrame 進行就地修改，不會傳回額外的 DataFrame 拷貝（copy）。接下來要使用的 append() 方法則不會修改原 DataFrame，相反地，它將傳回已添加新列的 DataFrame 拷貝。

append() 的第一個參數必須是另一個 DataFrame、Series、字典，或包含這幾類資料的串列。讓我們嘗試將一個字典傳給 append()，看看會發生什麼事：

In

```
names = pd.read_csv('data/names.csv')
names.append({'Name':'Aria', 'Age':1})
```

重新讀入原始的資料集

Out

```
Traceback (most recent call last):
...
TypeError: Can only append a Series if ignore_index=True or if the
Series has a name
```

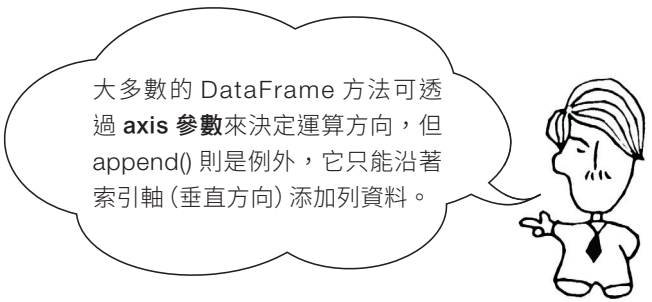
07 此處的錯誤訊息似乎不太正確，因為我們傳遞的是字典而非 Series。無論如何，它提示了我們該如何進行更正，即將 append() 的 **ignore_index 參數**設為 True。如此一來，我們便可將沒有列名（即索引標籤）的字典添加到 DataFrame 中：

In

```
names.append({'Name':'Aria', 'Age':1}, ignore_index=True)
```

Out

	Name	Age
0	Cornelia	70
1	Abbas	69
2	Penelope	4
3	Niko	2
4	Aria	1



08 雖然程式可成功運行，但將 ignore_index 參數設為 True 時，會同時刪除 DataFrame 的原始索引並替換成從 0 到 n-1 的 RangeIndex。為了證明這一點，讓我們先為 names 指定索引：

In

```
names.index = ['Canada', 'Canada', 'USA', 'USA']
names
```

Out

	Name	Age
Canada	Cornelia	70
Canada	Abbas	69
USA	Penelope	4
USA	Niko	2

09 重新執行步驟 7 的程式碼後，我們會得到相同的結果。可以發現，剛剛設定的索引已替換成 RangeIndex 了。

In

```
names.append({'Name':'Aria', 'Age':1}, ignore_index=True)
```

Out

	Name	Age
0	Cornelia	70
1	Abbas	69
2	Penelope	4
3	Niko	2
4	Aria	1

10 為了解決以上問題，此處改成將一個指定了 **name 參數**的 Series 傳給 append()。該做法可以讓我們保留 DataFrame 中的原始索引，而 name 參數的值就會是新的列資料之索引標籤：

In

```
s = pd.Series({'Name': 'Zach', 'Age': 3}, name=len(names))
s
```

編註：步驟 9 的 append() 不會更改原始的 DataFrame，故此處的 names 代表步驟 8 所輸出的 names，len(names) 則為 4

Out

```
Name    Zach
Age        3
Name: 4, dtype: object
```

In

```
names.append(s)
```

Out

```
          Name        Age
Canada    Cornelia    70
Canada    Abbas       69
USA       Penelope    4
USA       Niko        2
4         Zach        3
```

剛剛指定的 name 參數會是新資料列的索引標籤

11 append() 比 loc 屬性更靈活的原因在於，它可以同時添加多個列，只要傳入一個 Series 的串列即可：

In

```
s1 = pd.Series({'Name': 'Zach', 'Age': 3}, name=len(names))
s2 = pd.Series({'Name': 'Zayd', 'Age': 2}, name='USA')
names.append([s1, s2])
```

← 把 s1 和 s2 放進一個串列，並傳給 append()

Out

```
          Name        Age
Canada    Cornelia    70
Canada    Abbas       69
USA       Penelope    4
USA       Niko        2
4         Zach        3
USA       Zayd        2
```

12 從上面的例子可見，在新增列資料時，我們需要輸入所有欄位的名稱和其對應值。在處理欄位數很多的 DataFrame 時，這個過程就會變得非常複雜。接下來，讓我們檢視 2016 年的棒球資料集（包含 22 個欄位），看看如何在該 DataFrame 中新增列資料：

In

```
bball_16 = pd.read_csv('data/baseball16.csv')
bball_16
```

Out

	playerID	yearID	stint	...	SH	SF	GIDP
0	altuv...	2016	1	...	3.0	7.0	15.0
1	bregm...	2016	1	...	0.0	1.0	1.0
2	castr...	2016	1	...	1.0	0.0	9.0
3	corre...	2016	1	...	0.0	3.0	12.0
4	gatti...	2016	1	...	0.0	5.0	12.0
...	...	...	...	...	...	...	...
11	reedaj01	2016	1	...	0.0	1.0	1.0
12	sprin...	2016	1	...	0.0	1.0	12.0
13	tucke...	2016	1	...	0.0	0.0	2.0
14	valbu...	2016	1	...	3.0	2.0	5.0
15	white...	2016	1	...	0.0	2.0	6.0

13 該資料集中包含了 22 個欄位，如果想利用先前的方法添加列資料，很容易打錯欄位名稱或漏掉某個欄位。若要避免這些錯誤，可先任意選擇資料集中的單一列，並在傳回的 Series 後方串連 to_dict()。如此一來，我們便可得到以字典表示的**範例列**（example row）：

In

```
data_dict = bball_16.iloc[0].to_dict()
```

選取首列的資料做為範例列

```
data_dict
```

Out

```
{'playerID': 'altuvjo01',
 'yearID': 2016,
 'stint': 1,
 'teamID': 'HOU',
 'lgID': 'AL',
 'G': 161,
 'AB': 640,
 'R': 108,
 'H': 216,
 '2B': 42,
 '3B': 5,
 'HR': 24,
 'RBI': 96.0,
 'SB': 30.0,
 'CS': 10.0,
 'BB': 60,
 'SO': 70.0,
 'IBB': 11.0,
 'HBP': 7.0,
 'SH': 3.0,
 'SF': 7.0,
 'GIDP': 15.0}
```

14 利用**字典生成式**（dictionary comprehension）來清除 data_dict 中的值：將字串資料指定為空字串（' '），數值資料則指定為缺失值（np.nan）。最終生成的字典可做為輸入新資料時的模板：

In

```
                                    將字串資料指定為空字串
new_data_dict = {k: '' if isinstance(v, str) else
    np.nan for k, v in data_dict.items()}
new_data_dict
```

Out

```
{'playerID': '',
 'yearID': nan,
 'stint': nan,
 'teamID': '',
 'lgID': '',
 'G': nan,
 'AB': nan,
 'R': nan,
 'H': nan,
 '2B': nan,
 '3B': nan,
 'HR': nan,
 'RBI': nan,
 'SB': nan,
 'CS': nan,
 'BB': nan,
 'SO': nan,
 'IBB': nan,
 'HBP': nan,
 'SH': nan,
 'SF': nan,
 'GIDP': nan}
```

了解更多

在添加多列資料時，每次只新增單一列是相當不效率的。接下來，我們會說明添加列資料的不同策略，並比較它們之間的差別。先使用步驟 13 的 data_dict 建立 1000 列的新資料，將結果以 Series 串列的形式傳回並存成 random_data 變數：

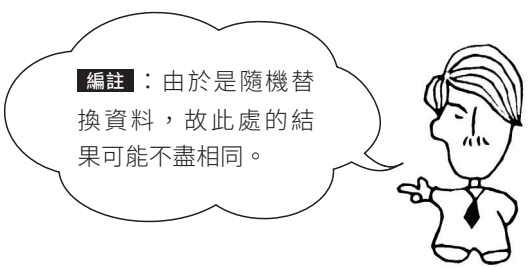
In

```
random_data = []
for i in range(1000):
    d = dict()
    for k, v in data_dict.items():
        if isinstance(v, str):
            d[k] = np.random.choice(list('abcde'))
        else:
            d[k] = np.random.randint(10)
    random_data.append(pd.Series(d, name=i + len(bball_16)))
random_data[0]
```

← 添加 1000 筆新資料
← 走訪 data_dict 中的項目
← 若是字串資料，則隨機替換為 a、b、c、d、e 中的一個字元
← 否則隨機替換為 0 到 10 之間的整數
← 將修改後的資料新增到 random_data 中
← 取得首列的資料

Out

```
playerID    d
yearID      0
stint       3
teamID      a
lgID        a
           ..
IBB         5
HBP         0
SH          3
SF          6
GIDP        3
Name: 16, Length: 22, dtype: object
```



我們先利用迴圈來**逐筆新增** random_data 中的資料，看看將其全部 1000 筆新資料添加至 DataFrame 中需花費多少時間：

In

```
%%timeit
bball_16_copy = bball_16.copy()
for row in random_data:
    bball_16_copy = bball_16_copy.append(row)
```

逐一走訪 random_data 中的資料
← 使用迴圈來新增列資料

```
Out
4.88 s ± 190 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

不過是添加 1000 列的資料，就耗費了近 5 秒鐘的時間。如果我們改為**傳入整個 Series 串列**，速度將得到巨大的提升：

```
In
%%timeit
bball_16_copy = bball_16.copy()
bball_16_copy = bball_16_copy.append(random_data)
```

```
Out
78.4 ms ± 6.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

如果傳入的是整個 Series 串列，則時間將縮短至小於 100ms（速度提升了超過 50 倍）。這是因為 Pandas 會先將 Series 串列轉換成單一的 DataFrame，然後再添加資料，從而大幅提升效能。

A.2 連接多個 DataFrame

concat() 函式可以將多個 DataFrame（或 Series）垂直或水平地連接在一起。按照慣例，當水平組合多個 Pandas 物件時，concat() 會先對齊每個物件的索引（編註：垂直連接時，則會先對齊欄位名稱）。

接下來，我們會使用 concat() 來水平及垂直地連接 DataFrame。同時，我們會嘗試更改其參數值來產生不同的結果。

動手做

01 讀入 2016 年和 2017 年的股票資料集，並將它們的股票代號 (Symbol 欄位) 設為索引：

In

```
stocks_2016 = pd.read_csv('data/stocks_2016.csv',index_col='Symbol')
stocks_2017 = pd.read_csv('data/stocks_2017.csv',index_col='Symbol')

stocks_2016
```

Out

Symbol	Shares	Low	High
AAPL	80	95	110
TSLA	50	80	130
WMT	40	55	70

In

```
stocks_2017
```

Out

Symbol	Shares	Low	High
AAPL	50	120	140
GE	100	30	40
IBM	87	75	95
SLB	20	55	85
TXN	500	15	23
TSLA	100	100	300

02 concat() 函式的第一個參數可接受一個串列，串列中存放的必須是 Pandas 物件（DataFrame 或 Series）。因此，我們先將 stocks_2016 和 stocks_2017 放到串列中，然後呼叫 concat() 函式將它們連接在一起。若沒有指定 concat() 的 **axis 參數**，代表 DataFrame 的欄位名稱會對齊，並沿著第 0 軸（即索引軸）進行垂直連接：

In

```
s_list = [stocks_2016, stocks_2017]
pd.concat(s_list)
```

Out

	Shares	Low	High
Symbol			
AAPL	80	95	110
TSLA	50	80	130
WMT	40	55	70
AAPL	50	120	140
GE	100	30	40
IBM	87	75	95
SLB	20	55	85
TXN	500	15	23
TSLA	100	100	300

stocks_2017 的資料連接在 stocks_2016 資料的下方

03 步驟 2 連接後的 DataFrame，並無法鑑別每筆資料的所屬年份。為此，我們可以使用 concat() 的 **keys 參數**來標記進行連接的 DataFrame。該標籤將出現在最外層的索引層級，並強制生成具有 MultiIndex 的 DataFrame。為了方便理解每個索引的意義，此處使用 **names 參數**來重新命名每個索引層級：

In

```
pd.concat(s_list, keys=['2016', '2017'], names=['Year', 'Symbol'])
```

Out

Year	Symbol	Shares	Low	High
2016	AAPL	80	95	110
	TSLA	50	80	130
	WMT	40	55	70
2017	AAPL	50	120	140
	GE	100	30	40
	IBM	87	75	95
	SLB	20	55	85
	TXN	500	15	23
	TSLA	100	100	300

04 此外，我們可以透過將 axis 參數改為 columns（或 1）來水平連接 DataFrame：

In

```
pd.concat(s_list, keys=['2016', '2017'], axis='columns', names=['Year', None])
```

Out

keys 參數指定的標籤會放在最外層的欄位名稱層級

Year	2016			2017		
	Shares	Low	High	Shares	Low	High
AAPL	80.0	95.0	110.0	50.0	120.0	140.0
TSLA	50.0	80.0	130.0	100.0	100.0	300.0
WMT	40.0	55.0	70.0	NaN	NaN	NaN
GE	NaN	NaN	NaN	100.0	30.0	40.0
IBM	NaN	NaN	NaN	87.0	75.0	95.0
SLB	NaN	NaN	NaN	20.0	55.0	85.0
TXN	NaN	NaN	NaN	500.0	15.0	23.0

小編補充 更改連接方向後，欄位名稱會變成 MultiIndex 物件，外部的欄位層級名稱為 Year，內部的欄位層級名稱則為 None（即沒有名稱）。

05 請注意，如果股票代號只存在於某一年份的資料，就會出現缺失值。預設情況下，concat () 會使用外部連接（outer join），即保留所有 DataFrame 的出現過的索引標籤。不過，我們也可只保留同時出現在 2016 年和 2017 年的股票資料。這種連接方式稱為內部連接（inner join），可透過將 concat () 的 **join 參數**改為 inner 來實現：

In

改為內部連接

```
pd.concat(s_list, join='inner', keys=['2016', '2017'],
          axis='columns', names=['Year', None])
```

Out

Year	2016			2017		
	Shares	Low	High	Shares	Low	High
Symbol						
AAPL	80	95	110	50	120	140
TSLA	50	80	130	100	100	300

只有這兩者在兩個年份中都有出現

了解更多

append() 是 concat() 的簡化版，它只能在 DataFrame 中添加新的列。事實上，append() 不過是在內部呼叫 concat() 函式，因此以下程式會產生與步驟 2 相同的結果：

In

```
stocks_2016.append(stocks_2017)
```

Out

Symbol	Shares	Low	High
AAPL	80	95	110
TSLA	50	80	130
WMT	40	55	70
AAPL	50	120	140
GE	100	30	40
IBM	87	75	95
SLB	20	55	85
TXN	500	15	23
TSLA	100	100	300

A.3 concat()、join() 和 merge() 的區別

DataFrame 的 merge() 和 join() 方法與 concat() 函式有許多相似的功能，都可用來組合 Pandas 物件。由於它們非常相似，而且在某些情況下可以相互替換，因此使用起來可能會令人非常困惑。為了搞懂它們之間的差異，請看下一頁的表格說明：

concat()	join()	merge()
Pandas 函式	DataFrame 方法	DataFrame 方法
可沿垂直或水平方向來組合多個 Pandas 物件	沿水平方向來組合多個 Pandas 物件	沿水平方向組合兩個 DataFrame
僅對齊索引	將原 DataFrame 的索引（或欄位）與另一物件的索引（而非欄位）對齊	將原 DataFrame 的欄位（或索引）與另一 DataFrame 的欄位（或索引）對齊
當同一物件的索引中出現重複項時會發生錯誤	當要連接的欄位（或索引）中出現重複項時，會透過執行笛卡兒積來合併資料	同 join()
預設為 outer join，但也可以將 join 參數改為 inner 來實現 inner join	預設為 left join，但也可以選擇 inner、outer 或 right join	預設為 inner join，但也可以選擇 left、outer 或 right join

小編補充 在組合時，join() 預設會對齊索引，merge() 則預設會對齊同名的欄位，二者均可用參數來改變對齊方式。另外關於 join 的概念，假設左資料為 {A:1, B:2, C:3}，右資料為 {A:5, B:6, D:7}，則各種 join 的結果如下：

- inner 為 {A:[1,5], B:[2,6]}
- outer 為 { A:[1,5], B:[2,6], C:[3,NaN], D:[NaN,7]}
- left 為 { A:[1,5], B:[2,6], C:[3,NaN]
- right 為 { A:[1,5], B:[2,6], D:[NaN,7]}。

接下來，我們會根據不同情況來組合 DataFrame。在第一種情況中，使用 concat() 會比較簡單，而第二種情況則推薦使用 merge()。

動手做

01 利用迴圈讀入 2016 年、2017 年和 2018 年的股票資料，進而傳回由 3 個 DataFrame 組成的串列：

In

利用串列生成式依序讀取名為 stocks_2016、stocks_2017 和 stocks_2018 的 CSV 檔案，並存成 stock_tables 串列

```
from IPython.display import display_html
years = 2016, 2017, 2018
stock_tables = [pd.read_csv(
    'data/stocks_{}.csv'.format(year), index_col='Symbol')
    for year in years]
stocks_2016, stocks_2017, stocks_2018 = stock_tables
stocks_2016
```

拆解 stock_tables 串列中的 3 個 DataFrame，並存成不同的變數

Out

Symbol	Shares	Low	High
AAPL	80	95	110
TSLA	50	80	130
WMT	40	55	70

In

```
stocks_2017
```

Out

Symbol	Shares	Low	High
AAPL	50	120	140
GE	100	30	40
IBM	87	75	95
SLB	20	55	85
TXN	500	15	23
TSLA	100	100	300

In

```
stocks_2018
```

Out

Symbol	Shares	Low	High
AAPL	40	135	170
AMZN	8	900	1125
TSLA	50	220	400

02 在 Pandas 的方法中，只有 concat() 函式能將 DataFrame 垂直組合在一起。讓我們將剛剛的 stock_tables 串列傳入 concat()，同時指定 keys 參數來標記不同的 DataFrame：

In

```
pd.concat(stock_tables, keys=years)
```

Out

		Shares	Low	High
	Symbol			
2016	AAPL	80	95	110
	TSLA	50	80	130
	WMT	40	55	70
2017	AAPL	50	120	140
	GE	100	30	40
	...	...	...	...
	TXN	500	15	23
	TSLA	100	100	300
2018	AAPL	40	135	170
	AMZN	8	900	1125
	TSLA	50	220	400

In

```
pd.concat(dict(zip(years, stock_tables)))
```

也可傳入字典（key 為 years，value 為 stock_tables）來得到相同的結果

小編補充 zip() 可將參數中對應的元素一一配對，例如 A 為 [1, 2]，B 為 [3, 4]，則 dict(zip(A, B)) = dict((1, 3), (2, 4)) = {1: 3, 2: 4}。

Out

		Shares	Low	High
	Symbol			
2016	AAPL	80	95	110
	TSLA	50	80	130
	WMT	40	55	70
2017	AAPL	50	120	140
	GE	100	30	40
	...	...	...	...
	TXN	500	15	23
	TSLA	100	100	300
2018	AAPL	40	135	170
	AMZN	8	900	1125
	TSLA	50	220	400

03 我們也可以更改 axis 參數來水平連接 DataFrame：

In

```
pd.concat(stock_tables, keys=[2016, 2017, 2018], axis='columns')
```

Out

	2016			...	2018		
	Shares	Low	High	...	Shares	Low	High
AAPL	80.0	95.0	110.0	...	40.0	135.0	170.0

	2016			...	2018		
	Shares	Low	High	...	Shares	Low	High
TSLA	50.0	80.0	130.0	...	50.0	220.0	400.0
WMT	40.0	55.0	70.0	...	NaN	NaN	NaN
GE	NaN	NaN	NaN	...	NaN	NaN	NaN
IBM	NaN	NaN	NaN	...	NaN	NaN	NaN
SLB	NaN	NaN	NaN	...	NaN	NaN	NaN
TXN	NaN	NaN	NaN	...	NaN	NaN	NaN
AMZN	NaN	NaN	NaN	...	8.0	900.0	1125.0

04 接下來，我們將依序使用 join() 和 merge() 來模擬 concat() 的功能。注意，我們需將 join() 的 **how 參數**設為 outer，以納入所有存在於 stock_2016 和 stock_2017 的列資料。由於 stock_2016 和 stock_2017 具有相同的欄位名稱，因此可再指定 lsuffix 或 rsuffix 來區分這些欄位：

In

```
stocks_2016.join(stocks_2017, lsuffix='_2016',rsuffix='_2017', how='outer')
```

左邊（stocks_2016）欄位名稱的尾端會加上 _2016

右邊（stocks_2017）欄位名稱的尾端會加上 _2017

Out

	Shares_2016	Low_2016	High_2016	Shares_2017	Low_2017	High_2017
Symbol						
AAPL	80.0	95.0	110.0	50.0	120.0	140.0
GE	NaN	NaN	NaN	100.0	30.0	40.0
IBM	NaN	NaN	NaN	87.0	75.0	95.0
SLB	NaN	NaN	NaN	20.0	55.0	85.0
TSLA	50.0	80.0	130.0	100.0	100.0	300.0
TXN	NaN	NaN	NaN	500.0	15.0	23.0
WMT	40.0	55.0	70.0	NaN	NaN	NaN

05 若想同時連接 stock_2018 的資料，可將由 DataFrame 組成的串列傳給 join()。儘管 join() 有 rsuffix 等參數來指定欄位名稱，但它僅在傳入單一 DataFrame 時有作用，無法處理 DataFrame 的串列。因此，我們先使用 add_suffix() 更改欄位名稱，然後才呼叫 join()：

In

```
other = [stocks_2017.add_suffix('_2017'),
         stocks_2018.add_suffix('_2018')]
stocks_2016.add_suffix('_2016').join(other, how='outer')
```

在 stocks_2017 中的欄位名稱尾端加上『_2017』

Out

	Shares_2016	Low_2016	High_2016	...	Shares_2018	Low_2018	High_2018
AAPL	80.0	95.0	110.0	...	40.0	135.0	170.0
TSLA	50.0	80.0	130.0	...	50.0	220.0	400.0
WMT	40.0	55.0	70.0	...	NaN	NaN	NaN
GE	NaN	NaN	NaN	...	NaN	NaN	NaN
IBM	NaN	NaN	NaN	...	NaN	NaN	NaN
SLB	NaN	NaN	NaN	...	NaN	NaN	NaN
TXN	NaN	NaN	NaN	...	NaN	NaN	NaN
AMZN	NaN	NaN	NaN	...	8.0	900.0	1125.0

06 讓我們檢查使用 concat() 和 join() 得到的結果是否相等：

In

```
stock_join = stocks_2016.add_suffix('_2016').join(other,how='outer')
stock_concat = pd.concat(dict(zip(years,stock_tables)),axis='columns')
level_1 = stock_concat.columns.get_level_values(1)
level_0 = stock_concat.columns.get_level_values(0).astype(str)
stock_concat.columns = level_1 + '_' + level_0
```

使用 join()

使用 concat()

編註：stock_concat 的欄位具有兩個層級（如步驟 3 輸出所示），這裡將 2 個層級的名稱以 '-' 連接在一起

```
stock_join.equals(stock_concat)
```

Out

```
True
```

07 現在，我們將改用 merge() 做到相同的事情。與 concat() 和 join() 不同，它只能組合兩個 DataFrame。預設情況下，merge() 會嘗試對齊每個 DataFrame 中，具有相同欄位名稱的值。不過，我們也可以將 **left_index** 和 **right_index 參數**設為 True 來改為對齊索引：

In

```
stocks_2016.merge(stocks_2017, left_index=True, right_index=True)
```

Out

小編補充 stocks_2016 和 stocks_2017 具有相同的欄位名稱，merge() 預設會在欄位名稱的尾端分別加上『_x』和『_y』來加以區分不同年份的資料。

	stocks_2016 的資料			stocks_2017 的資料		
	Shares_x	Low_x	High_x	Shares_y	Low_y	High_y
Symbol						
AAPL	80	95	110	50	120	140
TSLA	50	80	130	100	100	300

08 merge() 預設使用 inner join 並自動為同名的欄位加上後綴（suffix）。讓我們利用 how 參數把連接方式改為 outer join，然後對 2018 年的資料執行另一個 outer join 以模擬 concat() 的行為：

In

```
stock_merge = (stocks_2016
    .merge(stocks_2017, left_index=True,
           right_index=True, how='outer',
```

```
            suffixes=('_2016', '_2017')) ◀── 使用 suffixes 參數指定欄位名稱的後綴
    .merge(stocks_2018.add_suffix('_2018'),
            left_index=True, right_index=True,
            how='outer'))
stock_concat.sort_index().equals(stock_merge)
```

由於 merge() 產生的 DataFrame 之索引會經過排序，因此在進行相等性比較前，也要先排序 stock_concat 的索引

Out

```
True
```

09 接下來，我們將使用其它資料集（food_prices 和 food_transactions）做為示範。此處我們想對齊的並非索引標籤或欄位名稱，而是欄位中的值，這可以透過 merge() 來達成：

In

```
names = ['prices', 'transactions']
food_tables = [pd.read_csv('data/food_{}.csv'.format(name))
               for name in names]
food_prices, food_transactions = food_tables
food_prices ◀── 該資料集存有不同店鋪中，個別商品的單價
```

Out

	item	store	price	Date
0	pear	A	0.99	2017
1	pear	B	1.99	2017
2	peach	A	2.99	2017
3	peach	B	3.49	2017
4	banana	A	0.39	2017
5	banana	B	0.49	2017
6	steak	A	5.99	2017
7	steak	B	6.99	2017
8	steak	B	4.99	2015

In

```
food_transactions
```

← 該資料集存有不同店鋪中，顧客的交易記錄

Out

	custid	item	store	quantity
0	1	pear	A	5
1	1	banana	A	10
2	2	steak	B	3
3	2	pear	B	1
4	2	peach	B	2
5	2	steak	B	1
6	2	coconut	B	4

10 如果想找到每筆交易的總金額，需要用 item 和 store 欄位中的值來連接這些表格。為了清楚起見，我們使用 **on 參數**來指定欲進行連接的欄位名稱，但這個動作並不是必要的（**編註**：如果沒指定 on 參數，則會根據 DataFrame 的共同欄位來連接。以此例來說，food_transactions 和 food_prices 的共同欄位剛好就是 item 和 store 了）：

In

```
food_transactions.merge(food_prices, on=['item', 'store'])
```

Out

	custid	item	store	quantity	price	Date
0	1	pear	A	5	0.99	2017
1	1	banana	A	10	0.39	2017
2	2	steak	B	3	6.99	2017
3	2	steak	B	3	4.99	2015
4	2	steak	B	1	6.99	2017
5	2	steak	B	1	4.99	2015
6	2	pear	B	1	1.99	2017
7	2	peach	B	2	3.49	2017

← 同一年份中出現了兩筆 steak 的購買記錄

11 現在，price 與對應的 item 和 store 已正確對齊，但仍有問題需要解決。顧客 2（custid 為 2）共出現了 4 次 steak 的購買記錄，這是因為 store B 的 steak 項目在 food_prices 和 food_transactions 中都出現了兩次，因此會產生笛卡兒積並輸出 4 列的結果（編註：steak 在 food_prices 中有 2 筆，分別為 2015 和 2017 的價格）。另外，由於 food_prices 中不存在 coconut 項目的價格，因此步驟 10 的輸出缺少了這一個項目的資料。來嘗試解決這兩個問題：

In

```
food_transactions.merge(food_prices.query('Date == 2017'), how='left')
```

僅使用 2017 年的價格資料

使用 left join 來保留 food_transactions 中所有的交易資料

Out

	custid	item	store	quantity	price	Date
0	1	pear	A	5	0.99	2017.0
1	1	banana	A	10	0.39	2017.0
2	2	steak	B	3	6.99	2017.0
3	2	pear	B	1	1.99	2017.0
4	2	peach	B	2	3.49	2017.0
5	2	steak	B	1	6.99	2017.0
6	2	coconut	B	4	NaN	NaN

12 我們也可以使用 join() 來模擬以上結果，但必須先將 item 欄位和 store 欄位設定為 food_price 的索引：

In

```
food_prices_join = food_prices.query('Date == 2017') \
                              .set_index(['item', 'store'])
fod_prices_join
```

Out

		price	Date
item	store		
pear	A	0.99	2017
	B	1.99	2017
peach	A	2.99	2017
	B	3.49	2017
banana	A	0.39	2017
	B	0.49	2017
steak	A	5.99	2017
	B	6.99	2017

13 join() 固定會對齊右邊 DataFrame（以下列程式來說，即 food_prices_join）的索引，但左邊 DataFrame（即 food_transactions）則可對齊索引或欄位（預設對齊索引），若要改為欄位則需用 on 參數指定：

In

```
food_transactions.join(food_prices_join, on=['item', 'store'])
```

Out

	custid	item	store	quantity	price	Date
0	1	pear	A	5	0.99	2017.0
1	1	banana	A	10	0.39	2017.0
2	2	steak	B	3	6.99	2017.0
3	2	pear	B	1	1.99	2017.0
4	2	peach	B	2	3.49	2017.0
5	2	steak	B	1	6.99	2017.0
6	2	coconut	B	4	NaN	NaN

這裡的輸出與步驟 11 的結果相符。若要使用 concat() 模擬這個結果，需將 item 和 store 欄位移入這兩個 DataFrame 的索引中。不過在此例中，由於至少一個 DataFrame 中出現了重複的索引 (steak 和 B)，因此會產生錯誤：

In

```
pd.concat([food_transactions.set_index(['item', 'store']),
           food_prices.set_index(['item', 'store'])],
          axis='columns')
```

Out

```
Traceback (most recent call last):
...
ValueError: cannot handle a non-unique multi-index!
```

了解更多

即使不知道檔案名稱，我們也可以將資料夾中的特定檔案讀到 DataFrame 中。在 Python 提供的眾多檔案管理模組中，**glob 模組**是最受歡迎的。接下來使用的 gas prices 資料夾包含了 5 個不同的 CSV 文件，分別存有 2007 年起，各種等級天然氣的每週價格。在每個 CSV 檔案中只有兩個欄位：當週的起始日期與價格。

glob 模組中的 glob() 函式可接受檔案所在的路徑（以字串表示）。若想取得資料夾中的所有 CSV 檔案，可使用『*』來完成。在底下範例中，『*.csv』會傳回資料夾中以『.csv』結尾的所有檔案。glob() 函式所產生的結果為一串列（存有檔名字串），可以直接傳入 read_csv() 函式：

In

```
import glob
df_list = []
for filename in glob.glob('data/gas prices/*.csv'):
    df_list.append(pd.read_csv(filename, index_col='Week',
                               parse_dates=['Week']))
gas = pd.concat(df_list, axis='columns')
gas
```

走訪該路徑下，所有 CSV 檔案的名稱

讀入 CSV 檔案的資料（會存成 DataFrame），並放進 df_list 串列中

將 df_list 中的 DataFrame 以水平方式連接

Out

Week	All Grades	Diesel	Midgrade	Premium	Regular
2017-09-25	2.701	2.788	2.859	3.105	2.583
2017-09-18	2.750	2.791	2.906	3.151	2.634
2017-09-11	2.800	2.802	2.953	3.197	2.685
2017-09-04	2.794	2.758	2.946	3.191	2.679
2017-08-28	2.513	2.605	2.668	2.901	2.399
...	...	...	...	...	...
2007-01-29	2.213	2.413	2.277	2.381	2.165
2007-01-22	2.216	2.430	2.285	2.391	2.165
2007-01-15	2.280	2.463	2.347	2.453	2.229
2007-01-08	2.354	2.537	2.418	2.523	2.306
2007-01-01	2.382	2.580	2.442	2.547	2.334

A.4 連接到 SQL 資料庫

學習使用 SQL 是非常有用的，世界上大部分的資料都儲存在可接受 SQL 指令的資料庫中。在眾多的關聯式資料庫管理系統中，SQLite 是最受歡迎且易於使用的系統之一。

我們將探索 SQLite 所提供的 **chinook 資料庫**，其中包含 11 個表格（資料表），分別用來儲存音樂商店所需要的各種資料。在分析關聯式資料庫時，建議你先研究**資料庫圖**（database diagram，有時也稱為**實體關係圖**）以了解表格之間的關係：

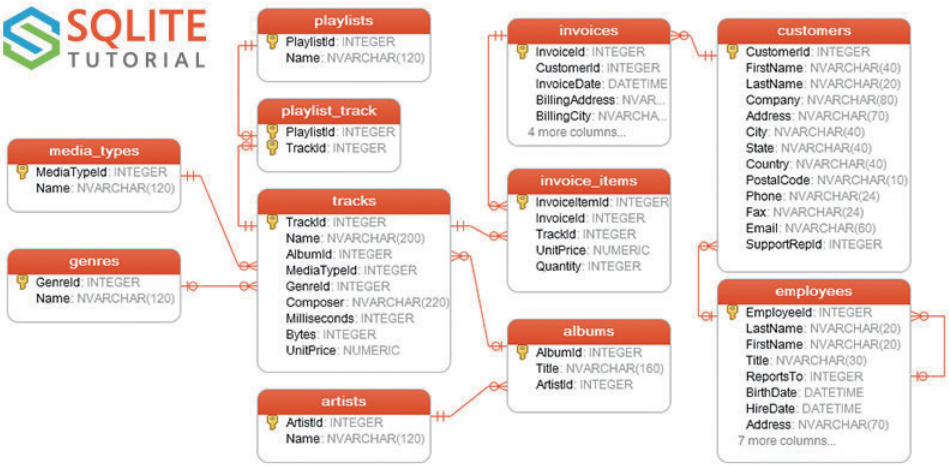


圖 A.1 chinook 資料庫的關係圖。

為了成功運行接下來的範例，需要安裝 Python 的 **sqlalchemy 套件**（相關的安裝程式已放在本章的 Jupyter Notebook 中），它是用來連接資料庫的首選工具。在這個例子中，你將學習如何連接到 SQLite 資料庫。接著我們會利用 merge() 來連接表格，以完成兩個查詢（query）工作。

動手做

01 從 chinook 資料庫讀取表格前，需先設定 SQLAlchemy 引擎。create_engine() 函式需要一個連接字串（connection string）才能正常運作。SQLite 的連接字串較簡單，就是以 'sqlite:///' 開頭，再加上資料庫的位置，如底下範例所示：

⚠ 其他的關聯式資料庫管理系統通常需要更複雜的連接字串，字串中需要提供用戶名稱、密碼、主機名、連接埠以及資料庫。此外，可能還需要提供 SQL 語言種類（dialect）與驅動程式名稱（driver）。連接字串的通用格式如下：dialect+driver://username:password@host:port/database。特定關聯式資料庫的驅動程式可能需要單獨安裝。

In

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///data/chinook.db')
```

設定引擎並連接到 chinook 資料庫

02 接下來，使用 read_sql_table() 函式讀入名為 tracks 的表格。該函式的第一個參數是表格名稱，第二個參數則是剛剛設定的 SQLAlchemy 引擎：

In

```
tracks = pd.read_sql_table('tracks', engine)
tracks
```

Out

	TrackId	Name	AlbumId	...	Milliseconds	Bytes	UnitPrice
0	1	For T...	1	...	343719	11170334	0.99
1	2	Balls...	2	...	342562	5510424	0.99
2	3	Fast ...	3	...	230619	3990994	0.99
3	4	Restl...	3	...	252051	4331779	0.99
4	5	Princ...	3	...	375418	6290521	0.99
...	...	...	...	...	...	...	...
3498	3499	Pini ...	343	...	286741	4718950	0.99
3499	3500	Strin...	344	...	139200	2283131	0.99
3500	3501	L'orf...	345	...	66639	1189062	0.99
3501	3502	Quint...	346	...	221331	3665114	0.99
3502	3503	Koyaa...	347	...	206005	3305164	0.99

03 現在，我們會根據圖 A.1 來回答幾個不同的查詢。首先，讓我們找出每首歌曲的風格類型和時間長度。若想連接 genres（另一個表格，存有歌曲的風格類型）和 tracks，可以使用 GenreId（genres 的主鍵，代表某一種歌曲風格的編號）：

⚠ 資料庫中的每個表格都有用來識別資料列的主鍵（primary key），在圖 A.1 中是用鑰匙圖形來表示。

In

```
(pd.read_sql_table('genres', engine) ← 讀入 genres 表格
     .merge(tracks[['GenreId', 'Milliseconds']], ← 與 tracks 表格中的 GenreId 和 Milliseconds 欄位合併
            on='GenreId', how='left')
     .drop('GenreId', axis='columns') ← 只保留歌曲風格的字串名稱，不需保留歌曲風格的編號
)
```

Out

歌曲的風格類型　歌曲的時間長度

	Name	Milliseconds
0	Rock	343719
1	Rock	342562
2	Rock	230619
3	Rock	252051
4	Rock	375418
...	...	...
3498	Class...	286741
3499	Class...	139200
3500	Class...	66639
3501	Class...	221331
3502	Opera	174813

04 現在，我們可以輕易地找到不同的風格類型中，歌曲的平均長度。為了方便解讀資料，讓我們把 Milliseconds 欄位的資料型別轉換為 timedelta 型別（編註：以便用 ' 時 : 分 : 秒 ' 的形式輸出）：

In

```
(pd.read_sql_table('genres', engine)
     .merge(tracks[['GenreId', 'Milliseconds']],
            on='GenreId', how='left')
     .drop('GenreId', axis='columns')
     .groupby('Name') ◄── 根據 Name 欄位（即歌曲的風格類型）來分組
     ['Milliseconds']
     .mean() ◄── 計算歌曲的平均時間長度
     .pipe(lambda s_: pd.to_timedelta(s_, unit='ms'))
                                        ↑
                                        將正確的度量單位以字串的方式傳入
     .dt.floor('s') ◄── 最小單位只保留到秒，如 00：02：14.25 會變成
     .sort_values()      00：02：14（dt.* 為 datetime 類的各種相關方法）
)
```

Out

```
Name
Rock And Roll      00:02:14
Opera              00:02:54
Hip Hop/Rap        00:02:58
Easy Listening     00:03:09
Bossa Nova         00:03:39
                     ...
Comedy             00:26:25
TV Shows           00:35:45
Drama              00:42:55
Science Fiction    00:43:45
Sci Fi & Fantasy   00:48:31
Name: Milliseconds, Length: 25, dtype: timedelta64[ns]
```

05 現在讓我們找出每位客戶花費的總金額。我們需要將 customers、invoice 和 invoice_items 表格連接起來，並利用 columns 參數取出需要的欄位：

In

```
cust = pd.read_sql_table('customers', engine,    ←— 讀入 customers 表格
    columns=['CustomerId','FirstName', 'LastName'])
invoice = pd.read_sql_table('invoices', engine, ←— 讀入 invoice 表格
    columns=['InvoiceId','CustomerId'])
ii = pd.read_sql_table('invoice_items', engine, ←— 讀入 invoice_items 表格
    columns=['InvoiceId', 'UnitPrice', 'Quantity'])
(cust.merge(invoice, on='CustomerId')
     .merge(ii, on='InvoiceId'))
```

← 將 3 個表格合併在一起

Out

	CustomerId	FirstName	LastName	InvoiceId	UnitPrice	Quantity
0	1	Luís	Gonça...	98	1.99	1
1	1	Luís	Gonça...	98	1.99	1
2	1	Luís	Gonça...	121	0.99	1
3	1	Luís	Gonça...	121	0.99	1
4	1	Luís	Gonça...	121	0.99	1
...	...	...	...	...	...	...
2235	59	Puja	Sriva...	284	0.99	1
2236	59	Puja	Sriva...	284	0.99	1
2237	59	Puja	Sriva...	284	0.99	1
2238	59	Puja	Sriva...	284	0.99	1
2239	59	Puja	Sriva...	284	0.99	1

06 我們現在可以將商品數量 (Quantity) 乘以商品單價 (UnitPrice)，並加總不同商品的花費，進而找出每位顧客的總花費金額。在底下程式的第 4 行，會用 assign() 來新增一個 Total 欄位以存放 Quantity 乘以 UnitPrice 的結果：

In

```
(cust
    .merge(invoice, on='CustomerId')
    .merge(ii, on='InvoiceId')
    .assign(Total=lambda df_:df_.Quantity * df_.UnitPrice)
    .groupby(['CustomerId', 'FirstName', 'LastName'])
    ['Total']
    .sum()
    .sort_values(ascending=False)
)
```

新增 Total 欄位

根據顧客的編號和姓名來分組

加總顧客在不同商品上的花費，即可得到每位顧客的消費金額

Out

```
CustomerId  FirstName  LastName
6           Helena     Holý          49.62
26          Richard    Cunningham    47.62
57          Luis       Rojas         46.62
46          Hugh       O'Reilly      45.62
45          Ladislav   Kovács        45.62
                                     ...
32          Aaron      Mitchell      37.62
31          Martha     Silk          37.62
29          Robert     Brown         37.62
27          Patrick    Gray          37.62
59          Puja       Srivastava    36.64
Name: Total, Length: 59, dtype: float64
```

了解更多

如果你精通 SQL，可以將 SQL 查詢以字串表示，並把它傳給 read_sql_query()。舉例來說，以下的程式碼可重現步驟 4 的輸出：

In

```
sql_string1 = '''
SELECT
    Name,
    time(avg(Milliseconds) / 1000, 'unixepoch') as avg_time
FROM (
      SELECT
          g.Name,
          t.Milliseconds
      FROM
          genres as g ◀── 用『g』表示 genres 表格
      JOIN
        用『t』表示 tracks 表格
              ↓
          tracks as t on
          g.genreid == t.genreid
     )
GROUP BY Name ◀── 根據歌曲的風格類型分組
ORDER BY avg_time''' ◀── 根據平均時間長度進行排序
pd.read_sql_query(sql_string1, engine)
```

Out

	Name	avg_time
0	Rock ...	00:02:14
1	Opera	00:02:54
2	Hip H...	00:02:58
3	Easy ...	00:03:09
4	Bossa...	00:03:39
...	...	...
20	Comedy	00:26:25
21	TV Shows	00:35:45
22	Drama	00:42:55
23	Scien...	00:43:45
24	Sci F...	00:48:31

要重現步驟 6 的結果，可使用以下 SQL 查詢：

In

```
sql_string2 = '''
   SELECT
         c.customerid,
         c.FirstName,
         c.LastName,
         sum(ii.quantity * ii.unitprice) as Total
   FROM
        customers as c
   JOIN
        invoices as i
        on c.customerid = i.customerid
   JOIN
       invoice_items as ii
       on i.invoiceid = ii.invoiceid
   GROUP BY
       c.customerid, c.FirstName, c.LastName
   ORDER BY
       Total desc'''

pd.read_sql_query(sql_string2, engine) ◀── 將上述 SQL 指令傳給 SQLite 進行查詢
```

Out

	CustomerId	FirstName	LastName	Total
0	6	Helena	Holý	49.62
1	26	Richard	Cunni...	47.62
2	57	Luis	Rojas	46.62
3	45	Ladislav	Kovács	45.62
4	46	Hugh	O'Reilly	45.62
...	...	...	...	...
54	53	Phil	Hughes	37.62
55	54	Steve	Murray	37.62
56	55	Mark	Taylor	37.62
57	56	Diego	Gutié...	37.62
58	59	Puja	Sriva...	36.64