前言

有一次，Noah 在海邊，一陣浪打到他身上，將他拉到更深的海裡，並且奪走他的呼吸。就在他恢復呼吸時，另一個浪又打到他身上。海浪消耗了他剩餘的大部分體力，並且將他拉入更深的海裡。就當他恢復呼吸，另一個浪還未到達時，他發現越是與浪和海掙搏，就消耗越多的體力。他不禁懷疑是否就此被海洋吞噬。他無法呼吸，而且身體疼痛。他被溺水嚇壞了。瀕臨死亡讓他更能專注於一件事，那就是保存他的體力，並且善用海浪，而不是與之為敵。

在一間並未實踐 DevOps 的新創公司就如同那天在那個沙灘所遭遇的情境一般。正式環境上的各種危機持續不間斷地上演著；所有操作都是手動的，告警成日地追逐著你，直到你的健康遭受迫害。唯一能協助你逃離這樣的死亡螺旋的方法就只有 DevOps 而已。

一件一件地做著對的事情直到一切清晰可見。首先，設置你的建置主機，開始測試你的程式碼，並且自動化手動的任務。著手一些可以重複至其他任務的工作，將這些首要之務做得正確，並且確保它被自動化。

一個在新創團隊或者任何公司常見的陷阱是找尋所謂的超級英雄。我們需要一個系統效能工程師，因為他們能夠解決效能的問題。我們需要首席營收官，因為他們能夠解決所有銷售的問題。我們需要 DevOps 工程師，因為它們能夠解決部署的流程。

在某一間公司，Noah 有一個延遲超過一年的專案，並且該網路應用程式已經用多種語言重寫了三次。這次的釋出僅需要系統效能工程師來完成。接著，我記得只有一位不知該說是勇敢還是愚鈍的工程師提問到：「什麼是系統效能工程師？」系統效能工程師是能夠讓所有東西大規模地運行而無虞。他明白在那個時候，他們需要一位超級英雄來拯救他們。超級英雄雇用症候群是在新產品或新創團隊中，用來撥亂反正的最好方法。但是，沒有任何一位工程師會救得了公司，除非他們先救得了自己。

Noah 也從其他的公司聽到類似的事情：「如果我們可以只雇用資深的 Erlang 工程師」，或者是「如果我們可以只雇用讓我們產生利潤的人」，或者是「如果我們可以只雇用能教會我們財務紀律的專家」，又或者是「如果我們能夠雇用一位 Swift 研發工程師」等，這樣的雇用模式是你所在的新創團隊或者是新產品所需要的最後一件事，還是需要的是了解那些只有超級英雄才能拯救的錯事，到底錯在哪裡。

在那個需要雇用系統效能工程師的公司案例裡，它最終發現問題是不適當的技術管理。事情責付給了錯誤的人（並且在聲量上，遮蔽了真正能解決這問題的人的聲音）。藉由移除低效的執行者、聆聽能夠完整解決問題的團隊成員、去除滿滿的工作列表、一次做一件對的事、和安插質檢管理，問題便可以在不需要超級英雄下被化解。

沒有任何人可以從你所在的新創團隊拯救你；你和你的團隊必須藉由激發良好的團隊合作、建立妥善的流程，並且對於你所處的組織抱持信心。這樣的問題的解決方案並非是透過雇用新的人員，而是透過誠實面對自己所處的境遇、思考如何導致這樣的狀況、並且一次只做一件對的事，直到你構築出屬於你自己的路。沒有任何的英雄，除非那英雄是你自身。

正如同在那風暴中的海上，並且緩慢地滅頂，沒有人會來拯救你或者是公司，除非自救。你正是你所處公司所需要的超級英雄，你或許會發現你的工作夥伴也是。

有一條遠離混亂的路，而這本書可以成為你的指引。讓我們開始吧！

對於作者來說，什麼是 DevOps 呢？

在軟體產業有許多抽象的概念難以被精準的定義。雲端運算、敏捷、和大數據正是這些隨著你所講述的對象不同而有不同定義的例子。與其嚴格地定義什麼是 DevOps，讓我們使用一些簡短的描述，呈現 DevOps 被實踐時的樣子：

- 開發團隊與維運團隊雙向的協作
- 維運任務的處理時間介於分鐘到小時之間，而非天到週之間。
- 開發者的充分參與；否則將回到開發與維運之間的對抗。
- 維運人員需要開發的技能，至少有 Bash 和 Python。
- 開發人員需要維運的技術，責任並非中止於完成程式碼的撰寫，而是包括著部署系統至正式環境以及監控。

- 自動化、自動化、自動化 : 你無法在沒有開發技能前提下，進行精確的自動化，並且你也無法在沒有維運的技能下，正確地進行自動化。
- 理想上，有自助服務提供給開發者，至少就部署而言。
- 可以透過 CI/CD 的流水線完成。
- GitOps。
- 開發與維運在任何方面上的雙向交流（比方說，工具、知識等）。
- 在設計、實作與部署裡，保持協作。就如同自動化，它無法在沒有分工合作下被成功地完成。
- 如果事情未被自動化，那麼它就會壞掉。
- 文化上，階層 < 流程。
- 微服務 > 單體服務。
- 持續部署系統正是軟體團隊的核心與靈魂。
- 沒有超級英雄。
- 持續交付是個必要選項，而非可選項。

如何使用本書

這本書以任意順序閱讀都是十分有幫助的。你可以隨意翻閱至感興趣的章節，並且應該能夠找到可以應用到實際工作中有用的事情。如果你是一位有經驗的 Python 程式設計師，你或許會想要大致看過第一章。同樣地，如果你對於實戰的故事、案例研究和專訪感興趣，你也許想從第十六章開始閱讀。

概念性主題

這本書的內容被分成七個概念性主題。第一群為 Python 基礎，它包括了對於語言和訊息自動化、命令列工具、和檔案系統自動化的簡潔介紹。

接著是維運，它囊括了有用的 Linux 工具、套件管理、建置系統、監控與控制工具、和自動化測試。這些全都是成為稱職的 DevOps 實踐者的必要主題。

雲端技術則是下個主題，這將會有雲端運算、基礎設施即程式碼、Kubernetes、和無伺服器架構的章節。目前圍繞在軟體產業的一個危機是缺乏有足夠雲端技術經驗的人才。精通本章將能為你的薪資和職涯帶來立竿見影的額外好處。

緊接著的是資料主題。機器學習維運和資料工程均有從 DevOps 角度的相關探討。本書將利用一個從頭至尾的完整機器學習專案，帶你遍歷建置、部署和基於使用 Flask、Sklean、Docker 和 Kubernetes 的機器學習模型維運。

最後一個主題是第十六章的案例探討、專訪和關於 DevOps 的戰爭故事。本章節適合作為你床前閱讀的內容。

Python 基礎

- 第一章，以 Python 實踐 DevOps 的必需知識
- 第二章，檔案與檔案系統操作自動化
- 第三章，使用命令列

維運

- 第四章，有用的 Linux 工具
- 第五章，套件管理
- 第六章，持續整合與持續部署
- 第七章，監控與日誌收集
- 第八章，運用 pytest 於 DevOps

雲端技術基礎

- 第九章，雲端運算
- 第十章，基礎設施即程式碼
- 第十一章，容器技術：Docker 和 Docker Compose
- 第十二章，容器調度：Kubernetes
- 第十三章，無伺服器技術

資料

- 第十四章，MLOps 和機器學習工程
- 第十五章，資料工程

案例研究

- 第十六章，DevOps 戰爭故事與訪談

本書編排慣例

以下為本書使用的編排規則：

斜體字（*Italic*）

表示新名詞、超連結、電子郵件位址、檔名以及副檔名。中文以楷體表示。

定寬字（`Constant width`）

用於程式原始碼，以及篇幅中參照到的程式元素，如變數、函式名稱、資料庫、資料型態、環境變數、程式碼語句以及關鍵字等。

定寬粗體字（**`Constant width bold`**）

代表使用者輸入的指令或文字。

定寬斜體字（*`Constant width italic`*）

代表需要配合使用者提供的變數，或者是使用者環境來更換的文字。

這個圖示代表提示或建議。

這個圖示代表一般的說明。

這個圖示代表警告或注意。

第二章

檔案與檔案系統操作自動化

Python 最具威力的特色之一就是它處理文字與檔案的能力。在 DevOps 的世界裡，無論是搜尋應用程式的日誌或者是派發組態檔案，都免不了持續地對檔案內的文字進行解析、搜索與更動。檔案是用來保存當前資料、程式與組態設定的工具；它們記錄著日誌資料以供查詢，也作為控制的媒介來描述需要進行控制的細節。透過 Python，你可以藉由重複利用的程式碼來創建、讀取和修改檔案與內容。將這些任務自動化的確是現代化 DevOps 有別於傳統系統管理的特點之一。你可以使用程式碼來進行一連串的操作，而不是透過一個需要手動執行的指令操作手冊。這樣的作法能減少操作上的疏失。如果你能夠有自信在每次執行系統時都採用相同的方式，你將能對整個操作流程更了解也更有自信。

讀取與寫入檔案

你可以利用 open 函式來建立一個關聯於檔案的物件，並且透過它來讀寫檔案。這個函式需要兩個引數，一個是代表檔案的位置，而另一個則是存取的模式（模式引數是可選的，預設為讀取）。除此之外，如果是文字或者是二進位檔案，可以使用模式來標明是要進行讀取或者是寫入。你可以基於模式 *r* 來開啟一個文字檔，並且取讀它的內容。這個檔案的物件提供一個 read 方法，來將檔案內容以字串的方式讀取出來：

```
In [1]: file_path = 'bookofdreams.txt'
In [2]: open_file = open(file_path, 'r')
In [3]: text = open_file.read()
In [4]: len(text)
Out[4]: 476909

In [5]: text[56]
Out[5]: 's'
```

```
In [6]: open_file
Out[6]: <_io.TextIOWrapper name='bookofdreams.txt' mode='r' encoding='UTF-8'>

In [7]: open_file.close()
```

在使用完檔案後，隨手關閉檔案是很好的習慣。Python 會在離開該檔案的執行範圍後，進行關閉來釋出資源。然而在釋出資源前，該資源都會被持續佔用，並且禁止其他程序對該檔案進行存取。

你也可以透過 `readlines` 方法來讀取檔案。該方法會利用檔案內的換行符號來將內容分割，並且傳回一個字串的串列。每一個字串都代表一行原始的內容：

```
In [8]: open_file = open(file_path, 'r')
In [9]: text = open_file.readlines()
In [10]: len(text)
Out[10]: 8796

In [11]: text[100]
Out[11]: 'science, when it admits the possibility of occasional hallucinations\n'

In [12]: open_file.close()
```

一個較為便利的開啟檔案方式是利用 `with` 語句。透過這個用法，使用者不需要再顯式地關閉檔案，取而代之的是 Python 會在結束程式執行區塊時，自動地將檔案關閉並且釋放資源：

```
In [13]: with open(file_path, 'r') as open_file:
    ...:     text = open_file.readlines()
    ...:

In [14]: text[101]
Out[14]: 'in the sane and healthy, also admits, of course, the existence of\n'

In [15]: open_file.closed
Out[15]: True
```

不同的作業系統會採用不同的斷行脫逸字元。以 Unix 系統來說，使用的是 `\n`；而 Windows 系統則是使用 `\r\n`。當開始檔案時，Python 會自動將這些脫逸字元統一轉換為 `\n`。如果你正開啟一個二進制的檔案，如 *.jpeg* 影像檔，卻用純文字的方式開啟，很可能就會因為這樣的轉換機制，進而使得檔案毀損。然而，當你讀取二進制檔案時，你可以藉由在模式引數後附加 *b*，來避免這樣的狀況發生：

```
In [15]: file_path = 'bookofdreamsghos00lang.pdf'
In [16]: with open(file_path, 'rb') as open_file:
    ...:     btext = open_file.read()
    ...:

In [17]: btext[0]
Out[17]: 37

In [18]: btext[:25]
Out[18]: b'%PDF-1.5\n%\xec\xf5\xf2\xe1\xe4\xef\xe3\xf5\xed\xe5\xee\xf4\n18'
```

加上這個字元到模式引數中，便不再對開啟檔案的斷行進行額外的轉換處理。

當要寫入檔案時，可以藉由引數 `w` 來使用寫入模式。`direnv` 是用來自動設定開發環境的工具。透過設置 *.envrc* 來定義環境變數與應用程式的執行環境；當你切換到有 *.envrc* 的資料夾時，`direnv` 會基於這個檔案來完成相關的設定。你可以利用 Python 並且採用寫入的旗標來開啟檔案，並在這個檔案中，將 `STAGE` 設為 `PROD`，以及將 `TABLE_ID` 設為 `token-storage-1234`：

```
In [19]: text = '''export STAGE=PROD
    ...: export TABLE_ID=token-storage-1234'''

In [20]: with open('.envrc', 'w') as opened_file:
    ...:     opened_file.write(text)
    ...:

In [21]: !cat .envrc
export STAGE=PROD
export TABLE_ID=token-storage-1234
```

請注意！函式庫 `pathlib` 的 `write` 方法會覆寫已經存在的檔案。

`open` 函式在開啟檔案不存在時，會建立檔案，而當開啟檔案已存在時，則會採取覆寫的動作。如果希望開啟原來存在的檔案，並且將新增內容接續在檔案後頭，則需要使用附加旗標 `a`。這個旗標會使得新增的內容接續於檔案末尾，並且維持原先的內容不變。如果正在寫入非純文字的內容（比方說 *.jpeg*），並且採用旗標 `w` 或 `a`，將很可能導致檔案的毀損。這個毀損發生的原因很可能是 Python 在寫入文字資料時，將斷行字元轉換為合於該運行環境的斷行字元。為了能夠寫入二進制的資料，你可以使用旗標 `wb` 或 `ab`，來確保檔案的安全。

第三章會深入 pathlib。針對檔案讀取與寫入，有兩個有用且便利的函式。pathlib 會隱藏對檔案物件的操作細節，並且完成使用者的請求。以下範例是用來從檔案中讀取純文字：

```
In [35]: import pathlib

In [36]: path = pathlib.Path(
             "/Users/kbehrman/projects/autoscaler/check_pending.py")

In [37]: path.read_text()
```

如果要讀取二進制資料，可以使用 path.read_bytes 方法。

當想要覆寫檔案或者建立新檔案時，有一些方法可以用來寫入純文字內容或者是二進制內容：

```
In [38]: path = pathlib.Path("/Users/kbehrman/sp.config")

In [39]: path.write_text("LOG:DEBUG")
Out[39]: 9

In [40]: path = pathlib.Path("/Users/kbehrman/sp")
Out[41]: 8
```

針對非結構化的純文字內容，透過使用檔案物件所提供的 read 和 write 函式通常就足夠了，但如果需要面對更為複雜的資料呢？ Javascript Object Notation（JSON）作為儲存簡單的結構化資料格式，被廣泛地使用在現代化的網路服務上。這種格式採用了兩種資料結構，一種是相似於 Python 字典（dict）的鍵值對映射，另一種則是相似於 Python 串列（list）的物件串列。在資料型態的支援上則提供了數值、字串、布林值和空值。AWS Identity and Access Management（IAM）網路服務讓使用者可以控制 AWS 上的資源存取的權限。這個服務便利用了 JSON 來定義存取原則，以下為該原則的範例：

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "service-prefix:action-name",
        "Resource": "*",
        "Condition": {
            "DateGreaterThan": {"aws:CurrentTime": "2017-07-01T00:00:00Z"},
            "DateLessThan": {"aws:CurrentTime": "2017-12-31T23:59:59Z"}
        }
    }
}
```

你能夠使用標準的檔案物件讀寫方法存取這類檔案的內容：

```
In [8]: with open('service-policy.json', 'r') as opened_file:
   ...:     policy = opened_file.readlines()
   ...:
   ...:
```

取決於採用的讀取方法，讀取出來的結果可能是單一字串或者是一個字串的串列，但都無法立即地被使用：

```
In [9]: print(policy)
['{\n',
 '    "Version": "2012-10-17",
\n',
 '    "Statement": {\n',
 '        "Effect": "Allow",
\n',
 '        "Action": "service-prefix:action-name",
\n',
 '        "Resource": "*",
\n',
 '        "Condition": {\n',
 '            "DateGreaterThan": {"aws:CurrentTime": "2017-07-01T00:00:00Z"},
\n',
 '            "DateLessThan": {"aws:CurrentTime": "2017-12-31T23:59:59Z"}\n',
 '        }\n',
 '    }\n',
 '}\n']
```

我們必須將讀出的字串進行解析，並且將它們恢復成原來的資料結構與型別，這將是個大工程。一個更好的做法是使用 json 函式模組：

```
In [10]: import json

In [11]: with open('service-policy.json', 'r') as opened_file:
    ...:     policy = json.load(opened_file)
    ...:
    ...:
    ...:
```

這個模組會解析 JSON 格式的資料，並且使用相應的 Python 資料結構將資料封裝後傳回：

```
In [13]: from pprint import pprint

In [14]: pprint(policy)
{'Statement': {'Action': 'service-prefix:action-name',
               'Condition': {'DateGreaterThan':
```

```
                                {'aws:CurrentTime': '2017-07-01T00:00:00Z'},
                            'DateLessThan':
                                {'aws:CurrentTime': '2017-12-31T23:59:59Z'}},
              'Effect': 'Allow',
              'Resource': '*'},
 'Version': '2012-10-17'}
```

pprint 函式模組能自動地把 Python 物件轉換成適合傾印出來的格式。輸出的內容將更容易被閱讀，同時也是一個用於檢視巢狀資料結構的便利方式。

現在，你可以基於檔案的原始結構來使用這些資料。用以下的例子來說明如何更動存取原則來管理 S3 資源：

```
In [15]: policy['Statement']['Resource'] = 'S3'

In [16]: pprint(policy)
{'Statement': {'Action': 'service-prefix:action-name',
               'Condition': {'DateGreaterThan':
                                {'aws:CurrentTime': '2017-07-01T00:00:00Z'},
                             'DateLessThan':
                                {'aws:CurrentTime': '2017-12-31T23:59:59Z'}},
               'Effect': 'Allow',
               'Resource': 'S3'},
 'Version': '2012-10-17'}
```

你可以利用 json.dump 方法將 Python 的字典內容輸出成 JSON 檔案：

```
In [17]: with open('service-policy.json', 'w') as opened_file:
    ...:     policy = json.dump(policy, opened_file)
    ...:
    ...:
    ...:
```

另一種常用於組態檔案的語言是 *YAML*（YAML 並不是一種標記語言）。它是 JSON 的擴展集，但透過使用類似於 Python 的空格寫法，使得本身的格式更為精巧。

Ansible 是一種用來處理軟體組態、管理和部署的自動化工具。Ansible 使用一種名為 *playbook* 的檔案來定義需要自動化的動作，而 playbook 便是使用 YAML 格式：

```
---
- hosts: webservers
  vars:
    http_port: 80
```

```
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  ...
```

最常用來解析 YAML 檔案的函式庫就是 PyYAML。它並不是 Python 的標準函式庫，因此你必須透過 pip 來自行安裝：

```
$ pip install PyYAML
```

安裝完成後，便可以使用 PyYAML 來匯入與匯出 YAML 資料：

```
In [18]: import yaml

In [19]: with open('verify-apache.yml', 'r') as opened_file:
    ...:     verify_apache = yaml.safe_load(opened_file)
    ...:
```

資料以熟悉的 Python 資料結構載入（一個含有字典的串列）：

```
In [20]: pprint(verify_apache)
[{'handlers': [{'name': 'restart apache',
                'service': {'name': 'httpd', 'state': 'restarted'}}],
  'hosts': 'webservers',
  'remote_user': 'root',
  'tasks': [{'name': 'ensure apache is at the latest version',
             'yum': {'name': 'httpd', 'state': 'latest'}},
            {'name': 'write the apache config file',
             'notify': ['restart apache'],
             'template': {'dest': '/etc/httpd.conf', 'src': '/srv/httpd.j2'}},
            {'name': 'ensure apache is running',
             'service': {'name': 'httpd', 'state': 'started'}}],
  'vars': {'http_port': 80, 'max_clients': 200}}]
```

你也可以將 Python 資料以 YAML 的格式進行儲存：

```
In [22]: with open('verify-apache.yml', 'w') as opened_file:
    ...:     yaml.dump(verify_apache, opened_file)
    ...:
    ...:
    ...:
```

另一種被廣泛用來表示結構性資料的語言是可擴展性標記語言（XML）。這種語言是由階層式的標籤元素所組成。過去有很多網站使用 XML 作為資料傳輸的格式，舉例來說，如簡易資訊聚合（Real Simple Syndication，RSS）。使用者利用 RSS 來追蹤以及獲得網站更新的訊息，另外，也可以使用它來追蹤來自不同處的文章刊登消息。Python 提供 `xml` 的函式庫來處理 XML 文件，它將 XML 文件中的階層式結構映射成一個樹狀的資料結構。樹的節點便是標記元素，而元素之間的父子關係則用來代表文件中元素的階層結構，最頂層的父節點便是整份文件的根節點。以下是解析 RSS 文件並且取得根節點的範例：

```
In [1]: import xml.etree.ElementTree as ET
In [2]: tree = ET.parse('http_feeds.feedburner.com_oreilly_radar_atom.xml')

In [3]: root = tree.getroot()

In [4]: root
Out[4]: <Element '{http://www.w3.org/2005/Atom}feed' at 0x11292c958>
```

你可以藉由迴圈從樹的頂端一路往下，遍尋所有的子節點：

```
In [5]: for child in root:
   ...:     print(child.tag, child.attrib)
   ...:
{http://www.w3.org/2005/Atom}title {}
{http://www.w3.org/2005/Atom}id {}
{http://www.w3.org/2005/Atom}updated {}
{http://www.w3.org/2005/Atom}subtitle {}
{http://www.w3.org/2005/Atom}link {'href': 'https://www.oreilly.com'}
{http://www.w3.org/2005/Atom}link {'rel': 'hub',
                                   'href': 'http://pubsubhubbub.appspot.com/'}
{http://www.w3.org/2003/01/geo/wgs84_pos#}long {}
{http://rssnamespace.org/feedburner/ext/1.0}emailServiceId {}
...
```

XML 提供了**命名空間**（利用標籤劃分資料群組）的功能。XML 使用定義於大括號內的命名空間作為標籤，加在子標籤前。如果你了解階層結構，便可以基於它們的關係鏈搜尋元素。另外，也可以提供一個定義了所有命名空間的字典，便於之後的查找：

```
In [108]: ns = {'default':'http://www.w3.org/2005/Atom'}
In [106]: authors = root.findall("default:entry/default:author/default:name", ns)

In [107]: for author in authors:
     ...:     print(author.text)
     ...:
Nat Torkington
VM Brasseur
```

```
Adam Jacob
Roger Magoulas
Pete Skomoroch
Adrian Cockcroft
Ben Lorica
Nat Torkington
Alison McCauley
Tiffani Bell
Arun Gupta
```

試算表是使用逗號作為分隔（CSV）來存放資料。你可以使用屬於自己的方式對這類資料進行處理，也可以使用 Python 的 csv 模組以更簡便的方式來讀取這些資料：

```
In [16]: import csv
In [17]: file_path = '/Users/kbehrman/Downloads/registered_user_count_ytd.csv'

In [18]: with open(file_path, newline='') as csv_file:
    ...:     off_reader = csv.reader(csv_file, delimiter=',')
    ...:     for _ in range(5):
    ...:         print(next(off_reader))
    ...:
['Date', 'PreviousUserCount', 'UserCountTotal', 'UserCountDay']
['2014-01-02', '61', '5336', '5275']
['2014-01-03', '42', '5378', '5336']
['2014-01-04', '26', '5404', '5378']
['2014-01-05', '65', '5469', '5404']
```

csv reader 物件透過迭代的方式逐行地讀取 *.csv* 資料，以便讓開發者可一次處理一行資料。透過這樣的方式來處理資料，在面對 *.csv* 的大檔案時特別有用，因為我們並不希望一次性地讀入大量資料到記憶體中。當然，你可能需要針對多筆資料的數個欄位進行運算，且剛好檔案也不是太大時，還是可以一次性地將所有資料載入記憶體中。

Pandas 是資料科學領域中重要的套件。它提供了近似於試算表一樣強而有力的資料儲存結構（pandas.DataFrame）。如果你所要進行統計分析的資料結構就如同表格一般，或者是你想要逐行逐列進行資料處理時，那麼 DataFrame 正是你需要的工具。由於它是第三方所提供的套件，因此需要透過 pip 安裝才能使用。在使用上，可以透過多種方式將資料匯入 DataFrame，但一個最常見的方式便是透過 *.csv* 檔案：

```
In [54]: import pandas as pd

In [55]: df = pd.read_csv('sample-data.csv')

In [56]: type(df)
Out[56]: pandas.core.frame.DataFrame
```

可以透過 head 方法來檢視 DataFrame 的首列資訊：

```
In [57]: df.head(3)
Out[57]:
   Attributes     open       high        low      close     volume
0     Symbols        F          F          F          F          F
1        date      NaN        NaN        NaN        NaN        NaN
2  2018-01-02  11.3007    11.4271    11.2827    11.4271   20773320
```

另外，可以透過 describe 方法來取得資料的統計資訊：

```
In [58]: df.describe()
Out[58]:
        Attributes    open      high    low     close     volume
count          357     356       356    356       356        356
unique         357     290       288    297       288        356
top     2018-10-18  10.402    8.3363   10.2    9.8111   36298597
freq             1       5         4      3         4          1
```

當然也可以透過中括號與欄位的名稱取得單欄的所有資訊：

```
In [59]: df['close']
Out[59]:
0            F
1          NaN
2      11.4271
3      11.5174
4      11.7159
        ...
352      9.83
353      9.78
354      9.71
355      9.74
356      9.52
Name: close, Length: 357, dtype: object
```

Pandas 還有許多用於分析與處理表格資料的方法，並且也有許多介紹用法的書籍。如果你需要進行資料分析，它無疑是個值得被關注的工具。

使用正規表達式搜尋字串

Apache HTTP 伺服器是相當普遍被用來作為網頁伺服器的開源工具。它可以透過組態設定日誌檔的格式。通用日誌格式（Common Log Format，CLF）是一種廣泛被使用的格式。有許多不同的日誌分析工具都支援這種日誌格式。以下為此種日誌的格式：

```
<IP Address> <Client Id> <User Id> <Time> <Request> <Status> <Size>
```

下面是基於這個格式的內容範例：

```
127.0.0.1 - swills [13/Nov/2019:14:43:30 -0800] "GET /assets/234 HTTP/1.0" 200 2326
```

第一章介紹了正規表達式與 Python 的 re 模組，所以讓我們試著使用這些工具從通用日誌格式中擷取資訊。建構正規表達式的一個小訣竅是一部分一部分地建構表達式。透過這種方式可以讓你逐步建構部分的表達式，而不用陷在為完整的表達式進行複雜的除錯任務中。你可以透過特定的群組名稱來建立一個可以取出 IP 位址資訊的正規表達式：

```
In [1]: line = '127.0.0.1 - rj [13/Nov/2019:14:43:30 -0800] "GET HTTP/1.0" 200'

In [2]: re.search(r'(?P<IP>\d+\.\d+\.\d+\.\d+)', line)
Out[2]: <re.Match object; span=(0, 9), match='127.0.0.1'>

In [3]: m = re.search(r'(?P<IP>\d+\.\d+\.\d+\.\d+)', line)

In [4]: m.group('IP')
Out[4]: '127.0.0.1'
```

你也可以建立一個正規表達式來取得時間：

```
In [5]: r = r'\[(?P<Time>\d\d/\w{3}/\d{4}:\d{2}:\d{2}:\d{2})\]'

In [6]: m = re.search(r, line)

In [7]: m.group('Time')
Out[7]: '13/Nov/2019:14:43:30'
```

當然也可以如下面範例一般，擷取出多個元素，包含 IP、使用者、時間和請求方法：

```
In [8]:  r = r'(?P<IP>\d+\.\d+\.\d+\.\d+)'

In [9]: r += r' - (?P<User>\w+) '

In [10]: r += r'\[(?P<Time>\d\d/\w{3}/\d{4}:\d{2}:\d{2}:\d{2})\]'

In [11]: r += r' (?P<Request>".+")'

In [12]:  m = re.search(r, line)

In [13]: m.group('IP')
Out[13]: '127.0.0.1'

In [14]: m.group('User')
Out[14]: 'rj'

In [15]: m.group('Time')
```

```
Out[15]: '13/Nov/2019:14:43:30'

In [16]: m.group('Request')
Out[16]: '"GET HTTP/1.0"'
```

針對日誌單行進行解析是個有趣的試驗，但卻不是特別的有用。然而，你也可以利用這個正規表達式作為基礎，設計一個可以從完整的日誌中取出所需資訊的表達式。舉個例子來說，我們想要搜尋 2019 年 11 月 8 日送出 GET 請求的 IP 位址。利用先前的表達式，基於這個需求進行如下的修改：

```
In [62]: r = r'(?P<IP>\d+\.\d+\.\d+\.\d+)'
In [63]: r += r'- (?P<User>\w+)'
In [64]: r += r'\[(?P<Time>08/Nov/\d{4}:\d{2}:\d{2}:\d{2} [-+]\d{4})\]'
In [65]: r += r' (?P<Request>"GET .+")'
```

使用 finditer 方法來處理日誌，並且將匹配內容中的 IP 位址傾印出來：

```
In [66]: matched = re.finditer(r, access_log)

In [67]: for m in matched:
    ...:     print(m.group('IP'))
    ...:
127.0.0.1
342.3.2.33
```

基於各式各樣的純文字資訊與正規表達式可以完成許多的任務。如果你並未因這樣複雜的工作感到畏縮，你將會發現它們是處理純文字資訊最有效的工具之一。

處理大型檔案

很多時候需要對非常大的檔案進行處理。如果檔案內的資料可以被逐行處理，那麼使用 Python 來處理這個任務將相當簡單。不採用在此之前所介紹的方法，一次性地載入整個檔案到記憶體中，你可以一次讀取一行資料並且進行處理，然後再取出下一筆資料。已經讀取的資料將會被 Python 的垃圾回收器（garbage collector）自動地從記憶體中移除，釋回到記憶體中。

Python 可以自動地配置並且釋放記憶體。垃圾回收器（garbage collector）是處理這個任務的其中一個工具。雖然很少需要手動地處理垃圾回收，但 Python 仍提供了 gc 套件來讓開發者可以直接進行控制。

當讀取一個從不同作業系統所創建的檔案時，不同的作業系統使用不同的斷行字元是件麻煩的事。Windows 所創建的檔案斷行時，除了 \n 字元外，會額外添加 \r 字元。然而這個額外的字元在 Linux 作業系統上，僅會被視為一個普通的文字。如果你有一個大型的檔案，而你想要基於目前的作業系統轉換斷行字元時，可以開啟一個檔案並且逐行地讀出，再寫入另一個新檔案，Python 會自動地處理這個斷行的字元：

```
In [23]: with open('big-data.txt', 'r') as source_file:
    ...:     with open('big-data-corrected.txt', 'w') as target_file:
    ...:         for line in source_file:
    ...:             target_file.write(line)
    ...:
```

值得一提的是，你可以巢化 with 語句，一次開啟兩個檔案，然後透過迴圈逐行讀取來源檔案。你也可以定義一個生成器來處理檔案的讀取，尤其是需要解析多個檔案，並且逐行讀取資料的時候：

```
In [46]: def line_reader(file_path):
    ...:     with open(file_path, 'r') as source_file:
    ...:         for line in source_file:
    ...:             yield line
    ...:

In [47]: reader = line_reader('big-data.txt')

In [48]: with open('big-data-corrected.txt', 'w') as target_file:
    ...:     for line in reader:
    ...:         target_file.write(line)
    ...:
```

當無法使用斷行字元作為分割資料的工具時，比方說大型的二進制檔案，你可以透過檔案物件的 read 方法傳入一次需讀取的位元數量來逐塊讀取資料。在檔案讀取完畢時，read 方法將會回傳空的字串：

```
In [27]: with open('bb141548a754113e.jpg', 'rb') as source_file:
    ...:     while True:
    ...:         chunk = source_file.read(1024)
    ...:         if chunk:
    ...:             process_data(chunk)
    ...:         else:
    ...:             break
    ...:
```