序

自從我撰寫《*Python Essential Reference*》以來，已經過了 20 多年。那時，Python 是一個小得多的語言，它的標準程式庫（standard library）自帶一組實用的「功能（batteries）」。當時它是你的大腦還容納得下的東西。《*Essential Reference*》反映了那個時代，它是一本小書，你可以帶著它在荒島上或秘密金庫裡寫一些 Python 程式碼。在隨後的三次修訂中，《*Essential Reference*》堅持著這一願景，也就是要成為一本精簡但完整的語言參考書，因為如果你要在度假時用 Python 寫些程式碼，為什麼不運用它全部的能力呢？

今日，距離上一版的問世已經過了十多年，Python 世界變得大不相同。Python 不再是一種小眾的利基語言（niche language），而是已經成為了世界上最流行的程式語言之一。Python 程式設計師也有豐富的資訊可供參考，以進階編輯器、IDE、notebooks、網頁等形式觸手可及。事實上，當你可能會想要的所有參考資料，幾乎都可以透過按下幾個鍵召喚到你眼前時，或許就沒有什麼必要去查閱參考書了。

若要說有什麼變化的話，這種資訊檢索的便利性和 Python 宇宙的規模帶來了一種不同的挑戰。如果你只是剛剛開始學習或需要解決一個新的問題，那麼要找出從哪裡著手，可能會讓人不知所措。要把各種工具的功能與核心語言本身區分開來也很困難。這類問題就是本書存在的理由。

《*Python* 精粹》是一本關於 Python 程式設計的書。它並不試圖記錄所有可能或已經在 Python 中達成過的事情。它的重點是介紹一個現代化的、經過整理彙集（或精煉過）的語言核心。這本書的靈感源自於多年來我向科學家、工程師和軟體專業人員教授 Python 的經驗。然而，它也是編寫軟體程式庫、將 Python 擅長領域推向極致，以及找出什麼東西最有用的過程中所生成的產物。

在大多數情況下，本書聚焦於 Python 程式設計本身。這包括製作抽象層的技巧、程式結構、資料、函式、物件、模組，等等，這些主題對於從事任何規模 Python 專案的程式設計師都很有幫助。可以透過 IDE 輕鬆獲得的純粹參考資料（例如函式清單、命令名稱、引數等）通常會被省略。我還做了一個刻意的選擇，即不描述快速變化的 Python 工具世界，也就是編輯器、IDE、部署工作等的相關事宜。

也許會有爭議的是，我通常不關注與大規模軟體專案的管理有關的語言功能。Python 有時被用在大型且嚴肅的事情上，由數百萬行程式碼所組成的那種。這種應用需要專門的工具、設計和功能。它們還涉及到委員會和會議，以及要對非常重要的事情做出的決策。對於這本小書來說，所有的這些都太多了。

但或許比較誠實的答案是，我並沒有用 Python 來撰寫這樣的應用程式，而你也不應該這樣做。至少不是作為一種業餘愛好來進行。

在寫書的過程中，對於不斷發展的語言功能總是要有一個分界線。本書是在 Python 3.9 的時代撰寫的。因此，它不包括計畫在以後版本中新增的一些主要功能，例如結構化的模式匹配（structural pattern matching）。那是另一個時間和地點的話題了。

最後，也很重要的是，我認為保持程式設計的趣味性是很關鍵的。我希望我的書不僅能幫助你成為有生產力的 Python 程式設計師，而且還能捕捉到一些啟發人們運用 Python 來探索星空、在火星上駕駛直升機，以及在後院用水炮噴灑松鼠的魔法。

致謝

我要感謝技術審閱者 Shawn Brown、Sophie Tabac 和 Pete Fein，感謝他們有益的評論。我還要感謝我長期合作的編輯 Debra Williams Cauley，感謝她在這個專案和過去的專案中所付出的努力。許多上過我課程的學生對本書所涉及的主題產生了重大影響，即便是間接的影響。最後，也很重要的是，我要感謝 Paula、Thomas 與 Lewis 對我的支持和愛。

關於作者

David Beazley 是《*Python Essential Reference, Fourth Edition*》（Addison-Wesley，2010）和《*Python Cookbook, Third Edition*》（O'Reilly，2013）的作者。他目前透過他的公司 Dabeaz LLC（`www.dabeaz.com`）教授進階的電腦科學課程。自 1996 年以來，他一直都在使用 Python，並撰寫、演講和教授相關的知識。

3

程式結構和流程控制

本章涵蓋程式結構（program structure）和流程控制（control flow）的細節。主題包括條件式（conditionals）、迴圈（looping）、例外（exceptions）和情境管理器（context managers）。

3.1 程式結構和執行

Python 程式的結構由一連串的述句（statements）所組成的。所有的語言功能，包括變數指定（variable assignment）、運算式（expressions）、函式定義（function definitions）、類別（classes）和模組匯入（module imports），都是與其他所有述句具有同等地位的述句，這意味著任何述句幾乎都可以放在程式的任何地方（雖然有某些述句，如 `return`，只能出現在函式內部）。舉例來說，這段程式碼在一個條件式中定義了兩個不同版本的函式：

```
if debug:
    def square(x):
        if not isinstance(x,float):
            raise TypeError('Expected a float')
        return x * x
else:
    def square(x):
        return x * x
```

載入原始碼檔案（source files）時，直譯器會以述句出現的順序來執行它們，直到沒有述句要執行為止。這種執行模型（execution model）適用於你作為主程式執行的檔案，也適用於經由 `import` 載入的程式庫檔案。

3.2 條件式執行

`if`、`else` 與 `elif` 述句控制條件式的程式碼執行（conditional code execution）。一個條件式述句（conditional statement）的一般格式是：

```
if expression:
    statements
elif expression:
    statements
elif expression:
    statements
...
else:
    statements
```

如果沒有要採取什麼動作，你可以省略一個條件式的 `else` 及 `elif` 子句。若是某個特定的子句不存在有述句，那就使用 `pass`：

```
if expression:
    pass            # 待處理：請實作
else:
    statements
```

3.3 迴圈與迭代

你使用 `for` 和 `while` 述句來實作迴圈（loops）。這裡有個例子：

```
while expression:
    statements

for i in s:
    statements
```

`while` 述句會執行一些述句，直到關聯的運算式被估算為假（false）為止。`for` 述句會迭代過 `s` 的所有元素，直到沒有更多的元素可用為止。`for` 述句適用於支援迭代的任何物件。這包括內建的序列型別（sequence types），如串列、元組和字串，但也包括實作了迭代器協定（iterator protocol）的任何物件。

在述句 `for i in s` 中，變數 `i` 被稱為迭代變數（iteration variable）。在迴圈的每次迭代中，它都會從 `s` 接收到一個新的值。迭代變數的範疇（scope）並非 `for` 述句私屬的。如果之前定義的某個變數有相同的名稱，那麼該值就會被覆寫。此外，迭代變數在迴圈完成後還會保留最後的值。

如果迭代產生的元素是大小相同的可迭代物件（iterables），你可以用這樣的述句將它們的值拆分（unpack）到個別的迭代變數中：

```
s = [ (1, 2, 3), (4, 5, 6) ]

for x, y, z in s:
    statements
```

在這個例子中，s 必須包含或產生可迭代物件，每個都有三個元素。在每次迭代中，變數 x、y 和 z 的內容會被指定為相應的可迭代物件之項目。雖然最常見的是在 s 為一個元組序列（sequence of tuples）時如此使用，但只要 s 中的項目是任何種類的可迭代物件，包括串列、產生器和字串，拆分動作（unpacking）都會是有效的。

有時，在進行拆分時，會使用一個用完即丟的變數（throw-away variable），例如 _。舉例來說：

```
for x, _, z in s:
    statements
```

在這個例子中，仍然會有一個值被放到 _ 變數中，但該變數的名稱暗示著它不怎麼有趣，或不會在後續的述句中使用。

如果一個可迭代物件所產生的項目有不同的大小，你可以使用通配符拆分（wildcard unpacking）來把多個值放到一個變數中。例如：

```
s = [ (1, 2), (3, 4, 5), (6, 7, 8, 9) ]

for x, y, *extra in s:
    statements           # x = 1, y = 2, extra = []
                         # x = 3, y = 4, extra = [5]
                         # x = 6, y = 7, extra = [8, 9]
                         # ...
```

在這個例子中，至少需要兩個值 x 和 y，但 *extra 會接收可能出現的任何其餘的值（extra values）。這些值永遠都會被放到一個串列中。最多只有一個帶星號的變數（starred variable）可以出現在單一次拆分中，但它可以出現在任何位置。因此，這兩種變體都是合法的：

```
for *first, x, y in s:
    ...

for x, *middle, y in s:
    ...
```

跑迴圈時，除了資料值之外，追蹤記錄數值索引（numerical index）有時也有用處。這裡有個例子：

```
i = 0
for x in s:
    statements
    i += 1
```

Python 提供了一個內建函式 `enumerate()`，它可被用來簡化這種程式碼：

```
for i, x in enumerate(s):
    statements
```

`enumerate(s)` 會創建一個迭代器，它會產生元組 `(0, s[0])`、`(1, s[1])`、`(2, s[2])` 等，以此類推。可以透過 `enumerate()` 的 `start` 關鍵字引數提供一個開始計數的不同起始值（starting value）：

```
for i, x in enumerate(s, start=100):
    statements
```

在這個例子中，`(100, s[0])`、`(101, s[1])` 等等的這種形式的元組會被產生出來。另一個常見的迴圈問題是平行迭代兩個或更多個可迭代物件，舉例來說，寫出一個迴圈在每次迭代從不同的序列拿取項目：

```
# s 和 t 是兩個序列
i = 0
while i < len(s) and i < len(t):
    x = s[i]     # 從 s 拿取一個項目
    y = t[i]     # 從 t 拿取一個項目
    statements
    i += 1
```

這段程式碼可以用 `zip()` 函式來簡化。例如：

```
# s 和 t 是兩個序列
for x, y in zip(s, t):
    statements
```

`zip(s, t)` 把可迭代物件 `s` 和 `t` 結合成一個可迭代物件，由元組 `(s[0], t[0])`、`(s[1], t[1])`、`(s[2], t[2])` 等所構成，若是長度不等，就會停止在 `s` 和 `t` 中最短的那個。`zip()` 的結果是一個迭代器（iterator），它會在被迭代時產生結果。如果你希望這結果被轉換為一個串列，就用 `list(zip(s, t))`。

要跳出一個迴圈，就使用 break 述句。舉例來說，這段程式碼會從一個檔案讀取文字行，直到遇到一個空的文字行為止：

```
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            break          # 一個空白行，停止讀取
        # 處理剝除後的文字行（stripped line）
...
```

要跳到一個迴圈的下次迭代（next iteration，即跳過迴圈主體剩餘的部分），就用 continue 述句。當反轉一個測試和再縮排另一層會讓程式內嵌得太深或不必要的複雜，這個述句就很有用。作為一個例子，下列迴圈會跳過一個檔案中所有的空白行：

```
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            continue      # 跳過空白行
        # 處理剝除後的文字行
        ...
```

break 與 continue 述句只適用於所執行的最內層迴圈（innermost loop）。如果你需要跳出一個深層內嵌（deeply nested）的迴圈結構，你可以使用一個例外（exception）。Python 並沒有提供某種「goto」述句。你也可以把 else 述句接附到迴圈構造上，如下列範例所示：

```
# for-else
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            break
        # 處理剝除後的文字行
        ...
    else:
        raise RuntimeError('Missing section separator')
```

一個迴圈的 else 子句只會在迴圈跑到完成時被執行。這要不是即刻發生（若是迴圈完全沒執行的話），就是發生在最後一次迭代之後。如果迴圈使用 break 述句提早終止了，其 else 子句就會被跳過。

迴圈的 `else` 子句主要的用例是在迭代過資料、但必須設定或檢查迴圈是否過早中斷的某種旗標（flag）或條件的程式碼中。舉例來說，如果你沒有使用 `else`，前一段程式碼可能必須以一個旗標變數（flag variable）來改寫如下：

```
found_separator = False

with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            found_separator = True
            break
        # 處理剝除後的文字行
        ...
    if not found_separator:
        raise RuntimeError('Missing section separator')
```

3.4 例外

例外（exceptions）表示錯誤（errors），並且會脫離一個程式正常的控制流程。例外是用 `raise` 述句來提出（raise）的。`raise` 述句的一般格式是 `raise Exception([value])`，其中 `Exception` 是例外型別（exception type），而 `value` 是一個選擇性的值，給出關於例外的具體細節。下面是一個例子：

```
raise RuntimeError('Unrecoverable Error')
```

要捕捉一個例外，就用 `try` 和 `except` 述句，如這裡所示：

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError as e:
    statements
```

當一個例外發生，直譯器就會停止執行 `try` 區塊中的述句，並尋找一個與所發生的例外型別相匹配的 `except` 子句。若有找到，控制權就會傳遞給那個 `except` 子句中的第一個述句。在 `except` 子句執行完畢後，控制權就會轉交給出現在整個 `try-except` 區塊之後的第一個述句繼續執行。

`try` 述句沒有必要匹配所有可能發生的例外。如果找不到匹配的 `except` 子句，那麼例外就會繼續傳播，並可能在他處能夠實際處理該例外的不同 `try-except` 區塊中被捕獲。就程式設計風格（programming style）而言，你應該只捕捉你的程式碼可以

實際恢復的那些例外。如果恢復是不可能的，那麼讓例外繼續傳播往往會是更好的做法。

如果一個例外在沒有被捕獲的情況下，一路上升到程式的頂層，直譯器就會以一個錯誤訊息來放棄執行。

如果 `raise` 述句單獨被使用，最後產生的例外會再次被提出。這只在處理先前提出的例外時會起作用。例如：

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError:
    print("Well, that didn't work.")
    raise       # 重新提出目前的例外
```

每個 `except` 子句都可以和 `as var` 修飾符（modifier）一起使用，這給出了一個變數的名稱，若有例外發生，該例外型別的一個實體就會被放到這個變數中。例外處理器（exception handlers）可以檢視這個值來找出關於例外原因的更多資訊。舉例來說，你可以使用 `isinstance()` 來檢查例外型別。

例外有一些標準的屬性（standard attributes），在需要對錯誤採取進一步動作的程式碼中可能很有用：

`e.args`

提出例外時提供的引數元組（tuple of arguments）。在大多數情況下，這會是一個單項元組（one-item tuple）帶著描述錯誤的一個字串。對於 `OSError` 例外，該值會是一個 2-tuple（雙項元組）或 3-tuple（三項元組），含有一個整數的錯誤號碼（error number）、字串錯誤訊息（error message）和一個選擇性的檔名（filename）。

`e.__cause__`

如果例外是在處理另一個例外時，作為回應所刻意提出的，那這就是前一個例外（previous exception）。關於鏈串的例外（chained exceptions），請參閱後續章節。

`e.__context__`

如果例外是在處理另一個例外的過程中被提出的，那這就是前一個例外。

`e.__traceback__`

與例外關聯的堆疊回溯物件（stack traceback object）。

用來存放一個例外值的變數只能在關聯的 `except` 區塊內取用。一旦控制離開了該區塊，那個變數就會變為未定義（undefined）的。例如：

```
try:
    int('N/A')              # 提出 ValueError
except ValueError as e:
    print('Failed:', e)

print(e)     # 失敗 -> NameError: 'e' 未定義。
```

可以使用多個 `except` 子句來指定多個例外處理區塊（exception-handling blocks）：

```
try:
    do something
except TypeError as e:
    # 處理型別錯誤（Type error）
    ...
except ValueError as e:
    # 處理值的錯誤（Value error）
    ...
```

單一個處理器（handler）可以捕捉多個例外型別，像這樣：

```
try:
    do something
except (TypeError, ValueError) as e:
    # 處理型別或值的錯誤
    ...
```

要忽略一個例外，就像下面這樣使用 `pass` 述句：

```
try:
    do something
except ValueError:
    pass              # 什麼都不做（聳肩）
```

默默忽略錯誤往往是很危險的，也是難以追查的失誤之來源。即使真的忽略了，通常比較明智的做法是選擇性地在某個記錄（log）或其他地方中回報錯誤，以便後續查看。

除了與程式退出相關的那些例外，若要捕捉所有的例外，就像這樣使用 Exception：

```
try:
    do something
except Exception as e:
    print(f'An error occurred : {e!r}')
```

捕捉所有的例外時，你應該非常仔細地向使用者報告準確的錯誤資訊。舉例來說，在前面的程式碼中，有一個錯誤訊息和關聯的例外值被列印出來。如果你沒有包含關於例外值的任何資訊，那麼要除錯那些因為你意想不到的理由而失敗的程式碼，就會變得非常困難。

try 述句也支援一個 else 子句，它必須跟在最後一個 except 子句的後面。如果 try 區塊中的程式碼沒有提出例外，那段程式碼就會被執行。這裡有個例子：

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError as e:
    print(f'Unable to open foo : {e}')
    data = ''
else:
    data = file.read()
    file.close()
```

finally 定義必定會執行的某個清理動作（cleanup action），不管 try-except 區塊中發生了什麼。這裡是一個例子：

```
file = open('foo.txt', 'rt')
try:
    # 做某些事情
    ...
finally:
    file.close()
    # 不管發生了什麼，檔案都會被關閉
```

finally 子句並非用來捕捉錯誤。取而代之，它用於永遠都必須執行的程式碼，無論是否發生錯誤。若沒有例外被提出，finally 子句中的程式碼將在 try 區塊中的程式碼之後立即執行。若有例外發生，首先會執行相應的那個 except 區塊（如果有的話），然後將控制權傳遞給 finally 子句的第一個述句。如果，在那段程式碼執行完畢後，仍有一個例外等待處理，那麼該例外將被重新提出，由另一個例外處理器（exception handler）來捕獲。

3.4.1 例外階層架構

處理例外的一個挑戰是管理你的程式中可能發生的大量例外。例如，光是內建的例外就有 60 多種。再加上標準程式庫的其他部分，就變成了數百種可能的例外。此外，通常沒有辦法輕易地事先確定任何部分的程式碼可能提出什麼樣的例外。例外並不會作為函式的呼叫特徵式（calling signature）的一部分被記錄起來，也沒有任何一種編譯器能驗證你程式碼中的例外處理是否正確。因此，例外處理有時會讓人覺得雜亂無章、沒有組織。

認識到例外透過繼承（inheritance）被組織成一個階層架構（hierarchy）是有幫助的。與其針對特定的錯誤，不如關注更普遍的錯誤種類，可能更為容易。舉例來說，考慮在容器（container）中查找值的時候可能出現的不同錯誤：

```
try:
    item = items[index]
except IndexError:      # 如果項目是一個序列，就提出
    ...
except KeyError:        # 如果項目是一個映射，就提出
    ...
```

與其編寫程式碼來處理兩個非常特定的例外，不如這樣做可能會更容易：

```
try:
    item = items[index]
except LookupError:
    ...
```

`LookupError` 是代表一個例外高階分組（higher-level grouping of exceptions）的一個類別。`IndexError` 和 `KeyError` 都繼承自 `LookupError`，所以這個 `except` 子句可以捕獲其中任何一個。然而，`LookupError` 並沒有廣泛到會包括與查找（lookup）無關的錯誤。

表 3.1 描述了最常見的內建例外類別。

`BaseException` 類別很少在例外處理中直接使用，因為它可以匹配所有可能的例外。這包括影響流程控制的特殊例外，如 `SystemExit`、`KeyboardInterrupt` 和 `StopIteration`。捕捉這些很少會是你想要的。取而代之，所有與程式有關的正常錯誤都繼承自 `Exception`。`ArithmeticError` 是所有數學相關錯誤的基礎，如 `ZeroDivisionError`、`FloatingPointError` 和 `OverflowError`。`ImportError` 是所有匯入相關錯誤的基礎。`LookupError` 是所有容器查找相關錯誤的基礎。`OSError` 是所有源

於作業系統和環境的錯誤之基礎。OSError 包含了與檔案、網路連線、權限、管線、逾時等廣泛的相關例外。ValueError 例外通常是在給了運算一個壞的輸入值時提出的。UnicodeError 是 ValueError 的一個子類別，將所有與 Unicode 相關的編解碼錯誤歸為一組。

表 3.1 例外種類

例外類別	描述
BaseException	所有例外的根類別
Exception	所有與程式有關的錯誤的基礎類別
ArithmeticError	所有數學相關錯誤的基礎類別
ImportError	匯入相關錯誤的基礎類別
LookupError	所有容器查找錯誤的基礎類別
OSError	所有系統相關錯誤的基礎類別。IOError 和 EnvironmentError 是別名
ValueError	與值有關的錯誤的基礎類別，包括 Unicode
UnicodeError	與 Unicode 字串編碼有關的錯誤的基礎類別

表 3.2 顯示了一些常見的內建例外，它們直接繼承自 Exception，但並不是更大的例外分組的一部分。

表 3.2 其他內建例外

例外類別	描述
AssertionError	失敗的 assert 述句
AttributeError	在一個物件上的屬性查找出錯
EOFError	檔案結尾
MemoryError	可恢復的記憶體用盡錯誤
NameError	在區域或全域命名空間中沒有找到名稱
NotImplementedError	未實作的功能
RuntimeError	通用的「發生了壞事」的錯誤
TypeError	運算套用到一個型別錯誤的物件
UnboundLocalError	在指定一個值前用了區域變數

3.4.2 例外和流程控制

通常情況下，例外是為處理錯誤而保留的。然而，有一些例外被用來改變控制流程。這些例外，如表 3.3 所示，直接繼承自 `BaseException`。

表 3.3 用於流程控制的例外

例外類別	描述
`SystemExit`	提出以表示程式退出
`KeyboardInterrupt`	當程式透過 Control-C 被中斷時提出
`StopIteration`	提出以表示迭代結束

`SystemExit` 例外是用來使程式刻意終止的。作為一個引數，你可以提供一個整數的退出碼（exit code）或者一個字串訊息。如果給了一個字串，它將被列印到 `sys.stderr`，程式將以 1 的退出碼終止。這裡有一個典型的例子：

```
import sys

if len(sys.argv) != 2:
    raise SystemExit(f'Usage: {sys.argv[0]} filename)

filename = sys.argv[1]
```

當程式收到一個 `SIGINT` 訊號（signal）時（通常是透過在終端機中按下 Control-C），`KeyboardInterrupt` 例外就會被提出。這個例外有點不尋常，因為它是非同步（asynchronous）的，也就是說，它幾乎可能在任何時間和程式中的任何述句發生。Python 的預設行為是在這種情況發生時單純終止。如果你想控制 `SIGINT` 的傳遞，可以使用 `signal` 程式庫模組（請參閱第 9 章）。

`StopIteration` 例外是迭代協定（iteration protocol）的一部分，是迭代結束的訊號。

3.4.3 定義新的例外

所有內建的例外都是以類別來定義的。要創建一個新的例外，需要建立一個繼承自 `Exception` 的新類別定義，比如下面的例子：

```
class NetworkError(Exception):
    pass
```

要使用你的新例外，請使用 raise 述句，如下所示：

```
raise NetworkError('Cannot find host')
```

提出一個例外時，以 raise 述句提供的選擇性值會被用作例外類別建構器（class constructor）的引數。大多數情況下，這是包含某種錯誤訊息的一個字串。然而，使用者定義的例外可以被寫成接受一個或多個例外值，如本例所示：

```
class DeviceError(Exception):
    def __init__(self, errno, msg):
        self.args = (errno, msg)
        self.errno = errno
        self.errmsg = msg

# 提出一個例外（多個引數）
raise DeviceError(1, 'Not Responding')
```

當你建立了一個重新定義 __init__() 的自訂例外類別時，重要的是將包含 __init__() 引數的一個元組指定給屬性 self.args，如前面所示。這個屬性是在列印例外回溯訊息（traceback messages）時使用。如果你不定義它，那麼錯誤發生時，使用者將無法看到關於例外的任何有用資訊。

例外可以使用繼承來組織成一個階層架構。舉例來說，前面定義的 NetworkError 例外可以作為各種更具體錯誤的基礎類別（base class）。下面是一個例子：

```
class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

def error1():
    raise HostnameError('Unknown host')

def error2():
    raise TimeoutError('Timed out')

try:
    error1()
except NetworkError as e:
    if type(e) is HostnameError:
        # 為這種錯誤進行特殊動作
        ...
```

在這種情況下，except NetworkError 子句可以捕獲從 NetworkError 衍生出來的任何例外。要找到被提出的具體錯誤型別，就用 type() 檢視執行值的型別。

3.4.4 鏈串的例外

有時，為了回應一個例外，你可能想提出一個不同的例外。要做到這一點，就要提出一個鏈串的例外（chained exception）：

```
class ApplicationError(Exception):
    pass

def do_something():
    x = int('N/A')    # 提出 ValueError

def spam():
    try:
        do_something()
    except Exception as e:
        raise ApplicationError('It failed') from e
```

如果發生未捕獲的 ApplicationError，你會得到包括兩個例外的一個訊息。比如說：

```
>>> spam()
Traceback (most recent call last):
  File "c.py", line 9, in spam
    do_something()
  File "c.py", line 5, in do_something
    x = int('N/A')
ValueError: invalid literal for int() with base 10: 'N/A'
```

上述例外是下面例外的直接原因：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c.py", line 11, in spam
    raise ApplicationError('It failed') from e
__main__.ApplicationError: It failed
>>>
```

如果你捕捉到一個 ApplicationError，所產生例外的 __cause__ 屬性將包含其他的例外。例如：

```
try:
    spam()
```

```
except ApplicationError as e:
    print('It failed. Reason:', e.__cause__)
```

如果你想提出一個新的例外而不包括其他例外的串鏈，可以像這樣從 None 提出一個錯誤：

```
def spam():
    try:
        do_something()
    except Exception as e:
        raise ApplicationError('It failed') from None
```

出現在 except 區塊中的程式設計錯誤也會導致鏈串的例外，但其運作方式略有不同。舉例來說，假設你有一些像這樣有錯的程式碼：

```
def spam():
    try:
        do_something()
    except Exception as e:
        print('It failed:', err)      # err 未定義（打錯字）
```

由此產生的例外回溯訊息略有不同：

```
>>> spam()
Traceback (most recent call last):
  File "d.py", line 9, in spam
    do_something()
  File "d.py", line 5, in do_something
    x = int('N/A')
ValueError: invalid literal for int() with base 10: 'N/A'
```

在處理上述例外的過程中，發生了另一個例外：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "d.py", line 11, in spam
    print('It failed. Reason:', err)
NameError: name 'err' is not defined
>>>
```

若有一個未預期的例外在處理另一個例外時被提出，__context__ 屬性（而非 __cause__）就會持有錯誤發生時正在處理的那個例外的資訊。比如說：

```
try:
    spam()
except Exception as e:
```

```
        print('It failed. Reason:', e)
        if e.__context__:
            print('While handling:', e.__context__)
```

在例外串鏈中，預期的例外和意外的例外之間有一個重要的區別。在第一個例子中，程式碼撰寫的方式有預見例外發生的可能性。例如，程式碼被明確地包裹在一個 `try-except` 區塊中：

```
try:
    do_something()
except Exception as e:
    raise ApplicationError('It failed') from e
```

在第二種情況下，`except` 區塊中存在一個程式設計錯誤：

```
try:
    do_something()
except Exception as e:
    print('It failed:', err)     # err 未定義
```

這兩種情況的區別是很微妙的，但也很重要。這就是為什麼例外鏈資訊被放在 `__cause__` 或 `__context__` 屬性中。`__cause__` 屬性是保留給當你預期有失敗的可能性時使用的。`__context__` 屬性在這兩種情況下都會被設置，但是對於在處理另一個例外時提出的未預期例外，它將是唯一的資訊來源。

3.4.5 例外回溯

例外有一個關聯的堆疊回溯（stack traceback），提供關於錯誤發生位置的資訊。這個回溯訊息儲存在例外的 `__traceback__` 屬性中。為了報告或除錯的目的，你可能想自己產生回溯訊息。可以使用 `traceback` 模組來實作這一目的。比如說：

```
import traceback

try:
    spam()
except Exception as e:
    tblines = traceback.format_exception(type(e), e, e.__traceback__)
    tbmsg = ''.join(tblines)
    print('It failed:')
    print(tbmsg)
```

在這段程式碼中，`format_exception()` 產生一個字串串列，其中包含 Python 通常會在回溯訊息中產生的輸出。作為輸入，你提供例外型別、值和回溯資訊。