chapter

11 常用內建模組

學習目標

- 處理日期與時間
- 認識日誌的使用
- 運用規則表示式
- 管理檔案與目錄
- URL 處理

11.1 日期與時間

大多數的開發者，對於日期與時間通常是漫不經心，使用著似是而非的方式處理，因此，在正式認識 Python 提供的時間處理 API 前，得先來瞭解一些時間、日期的時空歷史等議題，如此才會知道，時間日期確實是個很複雜的議題，而使用程式來處理時間日期，也不單只是使用 API 的問題。

11.1.1 時間的度量

想度量時間，得先有個時間基準，大多數人知道格林威治（Greenwich）時間，那麼就先從這個時間基準開始認識。

格林威治標準時間

格林威治標準時間（Greenwich Mean Time），經常簡稱 **GMT 時間**，一開始是參考自格林威治皇家天文台的標準太陽時間，格林威治標準時間的正午是太陽抵達天空最高點之時，由於後面將述及的一些源由，**GMT 時間常不嚴謹（且有爭議性）地當成是 UTC 時間**。

GMT 透過觀察太陽而得，然而地球公轉軌道為橢圓形且速度不一，本身自轉亦緩慢減速中，因而會有越來越大的時間誤差，現在 GMT 已不作為標準時間使用。

世界時

世界時（Universal Time, UT）是藉由觀測遠方星體跨過子午線（meridian）而得，這會比觀察太陽來得準確一些，西元 1935 年，International Astronomical Union 建議使用更精確的 UT 來取代 GMT，**在 1972 年導入 UTC 之前，GMT 與 UT 是相同的**。

國際原子時

雖然觀察遠方星體會比觀察太陽來得精確，不過 UT 基本上仍受地球自轉速度影響而有誤差。**1967 年定義的國際原子時（International Atomic Time,**

TAI），將秒的國際單位（International System of Units, SI）定義為銫（caesium）原子輻射振動 9192631770 周耗費的時間，時間從 UT 的 1958 年開始同步。

世界協調時間

由於基於銫原子振動定義的秒長是固定的，然而地球自轉會越來越慢，這會使得 TAI 時間持續超前基於地球自轉的 UT 系列時間，為了保持 TAI 與 UT 時間不要差距過大，因而提出了具有折衷修正版本的世界協調時間（Coordinated Universal Time），常簡稱為 **UTC**。

UTC 經過了幾次的時間修正，為了簡化日後對時間的修正，**1972 年 UTC 採用了閏秒（leap second）修正**（1 January 1972 00:00:00 UTC 實際上為 1 January 1972 00:00:10 TAI），確保 UTC 與 UT 相差不會超過 0.9 秒，加入閏秒的時間通常在 6 月底或 12 月底，由巴黎的 International Earth Rotation and Reference Systems Service 決定何時加入閏秒。

在撰寫本章的這個時間點，最近一次的閏秒修正為 2016 年 12 月 31 日，為第 27 次閏秒修正，當時 TAI 已超前 UTC 有 37 秒之長。

Unix 時間

Unix 系統的時間表示法，定義為 **UTC 時間 1970 年（Unix 元年）1 月 1 日 00:00:00 為起點而經過的秒數**，不考慮閏秒修正，用以表達時間軸上某一**瞬間（instant）**。

epoch

某個特定時間的起點，時間軸上某一瞬間。例如 Unix epoch 選為 UTC 時間 1970 年 1 月 1 日 00:00:00，不少發源於 Unix 的系統、平台、軟體等，也都選擇這個時間作為時間表示法的起算點，例如稍後要介紹的 `time.time()` 傳回的數字，也是從 1970 年（Unix 元年）1 月 1 日 00:00:00 起經過的秒數。

提示 »» 以上是關於時間日期的重要整理，足以瞭解後續 API 該如何使用，有機會的話，應該在維基百科上，針對方才談到的主題，詳細認識時間與日期。

就以上這些說明來說有幾個重點：

- 就目前來說，即使標註為 GMT（無論是文件說明，或者是 API 的日期時間字串描述），實際上談到時間指的是 UTC 時間。
- 秒的單位定義是基於 TAI，也就是銫原子輻射振動次數。
- UTC 考量了地球自轉越來越慢而有閏秒修正，確保 UTC 與 UT 相差不會超過 0.9 秒。最近一次的閏秒修正為 2016 年 12 月 31 日，當時 TAI 實際上已超前 UTC 有 37 秒之長。
- Unix 時間是 1970 年 1 月 1 日 00:00:00 為起點而經過的秒數，不考慮閏秒，不少發源於 Unix 的系統、平台、軟體等，也都選擇這個時間作為時間表示法的起算點。

11.1.2 年曆與時區簡介

度量時間是一回事，表達日期又是另一回事，前面談到時間起點，都是使用公曆，中文世界又常稱為陽曆或西曆，在談到公曆之前，得稍微往前談一下其他曆法。

儒略曆

儒略曆（Julian calendar）是現今西曆的前身，用來取代羅馬曆（Roman calendar），於西元前 46 年被 Julius Caesar 採納，西元前 45 年實行，約於西元 4 年至 1582 年之間廣為各地採用。**儒略曆修正了羅馬曆隔三年設置一閏年的錯誤，改採四年一閏**。

格里高利曆

格里高利曆（Gregorian calendar）改革了儒略曆，由教宗 Pope Gregory XIII 於 1582 年頒行，**將儒略曆 1582 年 10 月 4 日星期四的隔天，訂為格里高利曆 1582 年 10 月 15 日星期五**。

不過**各個國家改曆的時間並不相同**，像英國、大英帝國（包含現今美國東部）改曆的時間是在 1752 年 9 月初，因此在 Unix/Linux 中查詢 1752 年月曆，會發現 9 月平白少了 11 天。

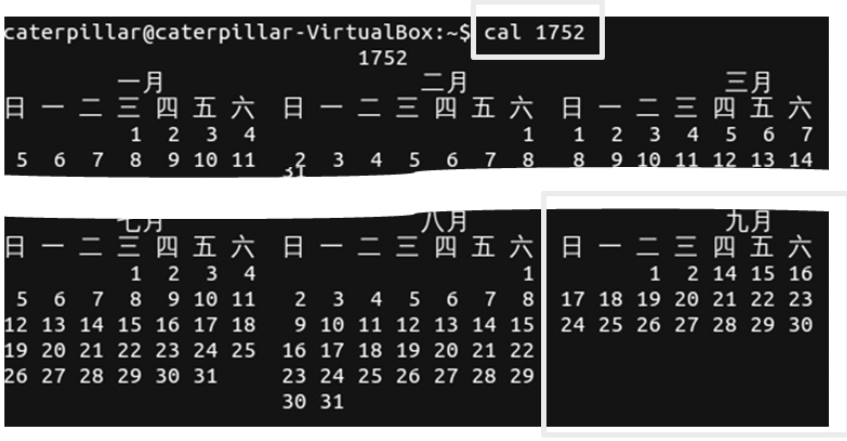

圖 11.1 Linux 中查詢 1752 年月曆

ISO8601 標準

在一些相對來說較新的時間日期 API 應用場合中，你可能會看過 ISO8601，**嚴格來說 ISO8601 並非年曆系統，而是時間日期表示方法的標準，用以統一時間日期的資料交換格式**，像是 yyyy-mm-ddTHH:MM:SS.SSS、yyyy-dddTHH:MM:SS.SSS、yyyy-Www-dTHH:MM:SS.SSS 之類的標準格式。

ISO8601 在資料定義上大部份與格里高利曆相同，因而有些處理時間日期資料的程式或 API，為了符合時間日期資料交換格式的標準，會採用 ISO8601。不過還是有些輕微差別。像是**在 ISO8601 的定義中，19 世紀是指 1900 至 1999 年（包含該年），而格里高利曆的 19 世紀是指 1801 年至 1900 年（包含該年）**。

時區

至於時區（Time zones），也許是各種時間日期的議題中最複雜的，每個地區的標準時間各不相同，因為這牽涉到地理、法律、經濟、社會甚至政治等問題。

從地理上來說，由於地球是圓的，基本上一邊白天另一邊就是夜晚，為了讓人們對時間的認知符合作息，因而設置了 **UTC 偏移（offset）**，大致上來說，經度每 15 度是偏移一小時，考量了 UTC 偏移的時間表示上，通常會在時間的最後標識 Z 符號。

不過有些國家的領土橫跨的經度很大，一個國家有多個時間反而造成困擾，不一定採取每 15 度偏移一小時的作法，像美國僅有四個時區，而中國、印度只採單一時區。

除了時區考量之外，有些高緯度國家，夏季、冬季日照時間差異很大，為了節省能源會儘量利用夏季日照，因而實施**日光節約時間（Daylight saving time）**，也稱為**夏季時間（Summer time）**，基本上就是在實施的第一天，讓白天的時間增加一小時，而最後一天結束後再調整一小時回來。

臺灣也曾實施過日光節約時間，後來因為效益不大而取消，臺灣現在許多開發者，多半不知道日光節約時間，因而會踩到誤區。舉例來說，臺灣 1975 年 3 月 31 日 23 時 59 分 59 秒的下一秒，是從 1975 年 4 月 1 日 1 時 0 分 0 秒開始。

如果得認真面對時間日期處理，認識以上的基本資訊是必要的，至少應該知道，一年的秒數絕對不是單純的 365 * 24 * 60 * 60，更不應該基於這類錯誤的觀念來進行時間與日期運算。

11.1.3 使用 time 模組

如果想獲取系統的時間，Python 的 time 模組提供了一層介面，用來呼叫各平台上的 C 程式庫函式，它提供的相關函式，通常與 epoch 有關。

time()、gmtime()與 localtime()

雖然大多數的平台都採取與 Unix 時間相同的 epoch，也就是 1970 年 1 月 1 日 00:00:00 為起點，不過，若想確定你的平台上的 epoch，也可以呼叫 time.gmtime(0)來確認。

```
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
>>>
```

gmtime()傳回了 struct_time 實例，是個具有 namedtuple 介面的物件，可以使用索引或屬性名稱來取得對應的年、月、日等數值，**採用的是 UTC**（雖然 gmtime()名稱上有 gmt 字樣），相關索引與屬性名稱會得到的值如下表所示：

表 11.1 struct_time 的索引與屬性

索引	屬性	值
0	tm_year	年，例如 2016
1	tm_mon	月，範圍 1 到 12
2	tm_mday	日，範圍 1 到 31
3	tm_hour	時，範圍 0 到 23
4	tm_min	分，範圍 0 到 59
5	tm_sec	秒，範圍 0 到 61
6	tm_wday	範圍 0 到 6，星期一為 0
7	tm_yday	範圍 1 到 366
8	tm_isdst	目前時區是否處於日光節約時間，1 為是，0 為否，-1 為未知

tm_sec 的值為 0 到 61，60 是為了閏秒，61 是為了一些歷史性的因素而存在。time.gmtime(0)表示從 epoch 起算經過了 0 秒，如果不指定數字，表示取得目前的時間並傳回 struct_time 實例，目前的時間是指透過 time.time()取得從 epoch（使用 UTC）至目前經過的秒數。例如：

```
>>> time.gmtime()
time.struct_time(tm_year=2020, tm_mon=9, tm_mday=15, tm_hour=3, tm_min=44,
tm_sec=36, tm_wday=1, tm_yday=259, tm_isdst=0)
>>> time.gmtime(time.time())
time.struct_time(tm_year=2020, tm_mon=9, tm_mday=15, tm_hour=3, tm_min=44,
tm_sec=49, tm_wday=1, tm_yday=259, tm_isdst=0)
>>> time.time()
1600141499.387509
>>>
```

UTC 是絕對時間，與時區無關，也沒有日光節約時間的問題，因此 tm_isdst 的值是 0。**time.time()傳回的是浮點數**，取得的秒數精度是否能到秒以下的單位，要看系統而定。

簡單來說，**time 模組提供的是低階的機器時間觀點，也就是從 epoch 起經過的秒數，然而有些輔助函式，可以作些簡單的轉換，以便成為人類可理解的時間概念**，除了 gmtime()可取得 UTC 時間之外，localtime()可提供目前所在時區

的時間，同樣地，localtime()可指定從 epoch 起經過的秒數，不指定時表示取得目前的系統時間，傳回的是 struct_time 實例：

```
>>> time.localtime()
time.struct_time(tm_year=2020, tm_mon=9, tm_mday=15, tm_hour=11, tm_min=46,
tm_sec=14, tm_wday=1, tm_yday=259, tm_isdst=0)
>>> time.gmtime()
time.struct_time(tm_year=2020, tm_mon=9, tm_mday=15, tm_hour=3, tm_min=46,
tm_sec=47, tm_wday=1, tm_yday=259, tm_isdst=0)
>>>
```

看到了嗎？臺灣的時區與 UTC 差了 8 小時，臺灣已經不實施日光節約時間了，因此 tm_isdst 的值是 0。

剖析時間字串

如果有個代表時間的字串，想剖析為 struct_time 實例，可以使用 strptime()函式。例如：

```
>>> time.strptime('2020-05-26', '%Y-%m-%d')
time.struct_time(tm_year=2020, tm_mon=5, tm_mday=26, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=1, tm_yday=147, tm_isdst=-1)
>>>
```

strptime()的第一個參數指定代表時間的字串，第二個參數為格式設定，可用的格式設定可參考 time.strftime()的說明[1]。這是個從字串剖析而來的時間，未考量時區資訊，無法確定是否採取日光節約時間，因此 tm_isdst 的值為-1。

gmtime()、localtime()與 strptime()都可以傳回 struct_time 實例，相對地，如果有個 struct_time 物件，想轉成從 epoch 起經過之秒數，可以使用 mktime()。例如：

```
>>> d = time.strptime('2020-05-26', '%Y-%m-%d')
>>> time.mktime(d)
1590422400.0
>>>
```

[1] time.strftime()：docs.python.org/3/library/time.html#time.strftime

可以看到，UTC 時間的 1975 年 3 月 31 日 14 時 59 分 59 秒時，臺灣時區的時間是 1975 年 3 月 31 日 22 時 59 分 59 秒，對 UTC 時間加兩小時後，轉為臺灣時區的時間是正確的 1975 年 4 月 1 日 1 時 59 分 59 秒（而不是 1975 年 4 月 1 日 0 時 59 分 59 秒）。

提示 »» 如果需要像圖 11.1 那樣的 Unix 月曆表示，可以使用 `calendar` 模組[4]。

11.2 日誌

系統中有許多需要記錄的資訊，例如例外發生之後，有些例外必須浮現至使用者介面層就繼續傳播，而對於開發人員或系統人員才有意義的例外，可以記錄下來，那麼該記錄哪些資訊（時間、資訊產生處...）？用何種方式記錄（檔案、資料庫、遠端主機...）？記錄格式（主控台、純文字、XML...）？這些都是在記錄時值得考慮的要素，也不是單純使用 `print()`就能解決，這時候，可以使用 Python 提供的 `loggoing` 模組來進行日誌的任務。

11.2.1 簡介 Logger

使用日誌的起點是 `logging.Logger` 類別，**一般來說，一個模組只需要一個 Logger 實例，因此，雖然可以直接建構 Logger 實例，不過建議透過 logging.getLogging()來取得 Logger 實例。**例如：

```
import logging
logger = logging.getLogger(__name__)
```

呼叫 getLogger()時，可以指定名稱，相同名稱下取得的 Logger 會是同一個實例，通常會使用__name__，因為在模組中__name__就是模組名稱（若模組在套件之中，會包含套件名稱）。

提示 »» 如果直接使用 `python` 執行某個模組，那麼 `__name__` 的值會是'__main__'。

---

4 `calendar` 模組：docs.python.org/3/library/calendar.html

呼叫 getLogger()時可以不指定名稱，這時會取得根 Logger（root logger），這是什麼意思？這是因為 Logger 實例之間有父子關係，可以使用點「.」來作父子階層關係的區分，**父階層相同的 Logger，父 Logger 的組態相同**。例如若有個 Logger 名稱為'openhome'，則名稱'openhome.some'與'openhome.other'的 Logger，它們的父 Logger 組態都是'openhome'名稱的 Logger 組態。

因此，如果使用了套件來管理多個模組，想要套件中的模組在進行日誌時，都使用相同的父組態，可以在套件的__init__.py 檔案中撰寫：

```
import logging
logger = logging.getLogger(__name__)
# 其他 logger 組態設定
```

在初次 import 套件中某個模組時，會先執行__init__.py 中的程式，此時__name__會是套件名稱，例如，若套件名稱為 openhome，那麼__name__就會是'openhome'，接著執行被 import 的模組時，例如 some 模組，那麼模組中的__name__會是'openhome.some'，如此就建立了 Logger 之間的父子關係。

取得 Logger 實例之後，可以使用 log()方法輸出訊息，輸出訊息時可以使用 Level 的靜態成員指定訊息層級（Level）。例如：

logging_demo basic_logger.py

```
import logging

logger = logging.getLogger(__name__)
logger.log(logging.DEBUG, 'DEBUG 訊息')
logger.log(logging.INFO, 'INFO 訊息')
logger.log(logging.WARNING, 'WARNING 訊息')
logger.log(logging.ERROR, 'ERROR 訊息')
logger.log(logging.CRITICAL, 'CRITICAL 訊息')
```

執行結果如下：

```
WARNING 訊息
ERROR 訊息
CRITICAL 訊息
```

咦？怎麼只看到 logging.WARNING 以下的訊息？logging 中的 DEBUG、INFO、WARNING、ERROR、CRITICAL 代表著不同的日誌等級，它們的實際的值都是數字，分別為 10、20、30、40、50，還有個 NOTSET 的值是 0。

在上面的範例中，還未曾對取得的 Logger 實例做任何設定，因此使用根 Logger 的日誌等級，預設是只有值大於 30，也就是 WARNING、ERROR、CRITICAL 的訊息才會輸出。

Logger 實例本身有個 setLevel()可以使用，不過要記得，**Logger 有階層關係，每個 Logger 處理完自己的日誌動作後，會再委託父 Logger 處理**，就日誌等級這部份來說，若上頭的 logger 使用 setLevel()設定為 logging.INFO，那麼呼叫 logger.log(logging.INFO, 'INFO 訊息')時，雖然可以通過實例本身的日誌等級 ，然而繼續委託給父 Logger 處理時，因為父 Logger 組態還是 logging.WARNING，結果訊息還是不會輸出。

因此，在不調整父 Logger 組態的情況下，直接設定 Logger 實例，就只能設定為更嚴格的日誌層級，才會有實際的效用。例如：

logging_demo basic_logger2.py

```
import logging

logger = logging.getLogger(__name__)
logger.setLevel(logging.ERROR)

logger.log(logging.DEBUG, 'DEBUG 訊息')
logger.log(logging.INFO, 'INFO 訊息')
logger.log(logging.WARNING, 'WARNING 訊息')
logger.log(logging.ERROR, 'ERROR 訊息')
logger.log(logging.CRITICAL, 'CRITICAL 訊息')
```

執行結果如下：

```
ERROR 訊息
CRITICAL 訊息
```

如果想調整根 Logger 的組態，可以使用 logging.basicConfig()，例如，可以指定 level 參數來調整根 Logger 的日誌等級：

logging_demo basic_logger3.py

```
import logging

logging.basicConfig(level = logging.DEBUG)

logger = logging.getLogger(__name__)
logger.log(logging.DEBUG, 'DEBUG 訊息')
logger.log(logging.INFO, 'INFO 訊息')
logger.log(logging.WARNING, 'WARNING 訊息')
logger.log(logging.ERROR, 'ERROR 訊息')
logger.log(logging.CRITICAL, 'CRITICAL 訊息')
```

執行結果如下：

```
DEBUG:__main__:DEBUG 訊息
INFO:__main__:INFO 訊息
WARNING:__main__:WARNING 訊息
ERROR:__main__:ERROR 訊息
CRITICAL:__main__:CRITICAL 訊息
```

除了使用 Logger 的 log()方法指定日誌等級之外，還可以使用 debug()、info()、warning()、error()、critical()等便捷方法，除此之外，對於例外，Logger 實例上提供了 exception()方法，日誌等級使用 ERROR，程式碼的語意上比較明確。

11.2.2 使用 Handler、Formatter 與 Filter

根 Logger 的日誌訊息預設會輸出至 sys.stderr，也就是標準錯誤，如果想修改能輸出至檔案，可以使用 logging.basicConfig()指定 filename 參數。例如 logging.basicConfig(filename = 'openhome.log')，如果子 Logger 實例沒有設定自己的處理器，就會輸出至指定的檔案。

子 Logger 實例可以透過 addHandler()新增自己的處理器，舉例來說，若想設定子 Logger 能輸出至檔案，可以如下：

logging_demo handler_demo.py

```
import logging

logging.basicConfig(filename = 'openhome.log')

logger = logging.getLogger(__name__)
```

```
logger.addHandler(logging.FileHandler('errors.log'))
logger.log(logging.ERROR, 'ERROR 訊息')
```

在這個範例中，設定了 logging.basicConfig(filename = 'openhome.log')，也在子 Logger 上新增了 FileHandler，謹記子 Logger 處理完日誌訊息之後，還會委託給父 Logger，因此若父 Logger 也有設定處理器，也會使用設定的處理器來處理日誌訊息，結果就是會看到 openhome.log 與 errors.log 兩個檔案。

在 logging 之中，提供了 StreamHandler、FileHandler 與 NullHandler，StreamHandler 可以指定輸出至指定的串流，像是 stderr、stdout。FileHandler 可以指定輸出至檔案，而 NullHandler 什麼都不做，有時在開發程式庫時並不是真的想輸出日誌，就可以使用它，除了這三個基本的處理器之外，更多進階的處理器，可以在 logging.handlers 模組[5]中找到，像是可與遠端機器溝通的 SocketHandler，支援 SMTP 的 SMTPHandler 等。

提示 »» logging.handlers 模組提供了大量的處理器實作，如果這仍不能滿足你，可以繼承 logging.Handler 或其他處理器類別來實作，實作方式可參考 logging.handlers 模組中的原始碼。

處理器在輸出訊息時，格式預設是使用指定的訊息，若要自訂格式，可以透過 logging.Formatter() 建立 Formatter 實例，再透過處理器的 setFormatter()設定 Formatter 實例。例如，顯示訊息時若想連帶顯示時間、Logger 名稱、日誌等級，可以如下：

logging_demo formatter_demo.py

```
import logging, sys

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler = logging.StreamHandler(sys.stderr)
handler.setFormatter(formatter)
```

5 logging.handlers 模組：docs.python.org/3/library/logging.handlers.html

```
logger = logging.getLogger(__name__)
logger.addHandler(handler)

logger.log(logging.ERROR, '發生了 XD 錯誤')
```

%(asctime)使用人類可理解的時間格式來顯示日誌的時間，%(name)顯示 Logger 名稱，%(levelname)顯示日誌層級，%(message)顯示指定的日誌訊息，除了這些格式設定之外，還有其他許多可用的設定，這部份是由 LogRecord 的屬性來定義，可直接參考官方線上文件[6]。

上面的範例執行結果如下所示：

```
2020-09-15 17:22:17,812 - __main__ - ERROR - 發生了 XD 錯誤
```

如果格式指定中出現了%(asctime)，內部會呼叫 formatTime()來進行時間的格式化，如果想控制時間的格式，可以在使用 logging.Formatter()時指定 datefmt 參數，指定的格式會使用 time.strftime()來進行格式化。

如果不喜歡%這個字元，可以在使用 logging.Formatter()時使用 style 參數指定其他字元。

如果除了使用 DEBUG、INFO、WARNING、ERROR、CRITICAL 等日誌層級過濾訊息是否輸出之外，還想要使用其他條件來過濾哪些訊息可以輸出，可以定義過濾器，你可以繼承 logging.Filter 類別並定義 filter(record)方法，或是定義一個物件具有 filter(record)方法，根據傳入的 LogRecord 取得日誌時的資訊，並傳回 0 決定不輸出訊息，傳回非 0 值決定輸出訊息。

不過 Python 3.2 以後，也可以使用函式作為過濾器了，Logger 或 Handler 實例都有 addFilter()方法，可以新增過濾器。以下是個簡單的範例：

logging_demo filter_demo.py

```
import logging, sys

logger = logging.getLogger(__name__)
logger.addFilter(lambda record: 'Orz' in record.msg)
```

6 LogRecord attributes：docs.python.org/3/library/logging.html#logrecord-attributes

```
logger.log(logging.ERROR, '發生了 XD 錯誤')
logger.log(logging.ERROR, '發生了 Orz 錯誤')
```

在這個範例中，針對某個字眼進行了過濾，只有訊息中包含'Orz'才會顯示出來：

```
發生了 Orz 錯誤
```

有關於 LogRecord 上可用的屬性，同樣可參考官方線上文件。

提示 »» 設定過濾器時別忘了，子 Logger 實例過濾後的訊息，還會再委託父 Logger，因此，若無法通過父 Logger 的過濾，訊息仍舊不會顯示。關於 Logger 進行日誌的整個流程，可以參考〈Logging Flow[7]〉。

11.2.3　使用 logging.config

以上都是使用程式撰寫方式，改變 Logger 物件的組態，實際上，可以透過 logging.config 模組，使用組態檔案來設定 Logger 組態，這很方便，例如程式開發階段，在組態檔案中設定 WARNING 等級的訊息就可輸出，在程式上線之後，若想關閉不會影響程式運行的警訊日誌，以減少程式不必要的輸出（不必要的日誌輸出會影響程式運行效率），只要在組態檔案中做個修改即可。

不過組態檔案的設定細節非常多而複雜，為了讓你有個起點，在 Python 的 logging 模組官方文件中有個範例，是使用設定 Logger 組態的參考範例，當中先舉了個使用程式組態的例子：

```
import logging

# 建立 logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# 建立主控台處理器並設定日誌等級為 DEBUG
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)
```

7 Logging Flow：docs.python.org/3/howto/logging.html#logging-flow

```
# 建立 formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s -
%(message)s')

# 將 formatter 設給 ch
ch.setFormatter(formatter)

# 將 ch 加入 logger
logger.addHandler(ch)

# 應用程式的程式碼
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

接著，這個程式被簡化了，相關組態資訊使用了 logging.config.fileConfig()從組態檔案讀取，不過那是舊式的方法，組態檔案的撰寫格式是ini，不方便也不易於閱讀，有興趣可以自行參考。

自 Python 3.2 開始，建議改用 logging.config.dictConfig()，可使用 dict 物件來設定組態資訊，這邊改寫官方的例子，改用 logging.config.dictConfig()，方才程式組態的內容被改寫為以下：

logging_demo config_demo.py

```
import logconf
import logging.config

logging.config.dictConfig(logconf.LOGGING_CONFIG)

# 建立 logger
logger = logging.getLogger('simple_example')

# 應用程式的程式碼
logger.debug('debug message')
logger.info('info message')
logger.warning('warn message')
logger.error('error message')
logger.critical('critical message')
```

其中 logconf 模組，就是一個撰寫了組態資訊的 logconf.py 檔案：

logging_demo logconf.py

```
LOGGING_CONFIG = {
    'version' : 1,
    'handlers' : {
        'console': {
            'class': 'logging.StreamHandler',
            'level': 'DEBUG',
            'formatter': 'simpleFormatter'
        }
    },
    'formatters': {
        'simpleFormatter': {
            'format': '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        }
    },
    'loggers' : {
        'simple_example' : {
            'level' : 'DEBUG',
            'handlers' : ['console']
        }
    }
}
```

被做為組態檔案的 logconf.py 本身就是 Python 原始碼，組態設定時使用的是 dict，最重要的是必須有個'version'鍵名稱，每個處理器、格式器、過濾器或 Logger 實例，都要有個名稱，以便設定時參考之用。dict 中可使用的鍵名稱，可參考〈Dictionary Schema Details[8]〉的說明。

若不想使用.py 作為設定檔，也可以使用 JSON，例如建立一個 logconf.json 檔案：

logging_demo logconf.json

```
{
    "version" : 1,
    "handlers" : {
        "console": {
            "class": "logging.StreamHandler",
```

8 Dictionary Schema Details ： docs.python.org/3/library/logging.config.html#dictionary-schema-details

```
            "level": "DEBUG",
            "formatter": "simpleFormatter"
        }
    },
    "formatters": {
        "simpleFormatter": {
            "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
        }
    },
    "loggers" : {
        "simple_example" : {
            "level" : "DEBUG",
            "handlers" : ["console"]
        }
    }
}
```

要留意的是，JSON 的規範中，必須使用雙引號來含括鍵名稱。有了這個 JSON 檔案，就可以運用 10.3.2 介紹的 `json.load()`來讀取 JSON，並作為 `logging.config.dictConfig()`的引數：

logging_demo config_demo2.py

```
import logging, json
import logging.config

with open('logconf.json') as config:
    LOGGING_CONFIG = json.load(config)
    logging.config.dictConfig(LOGGING_CONFIG)

# 建立 logger
logger = logging.getLogger('simple_example')

# 應用程式的程式碼
logger.debug('debug message')
logger.info('info message')
logger.warning('warning message')
logger.error('error message')
logger.critical('critical message')
```

11.3 規則表示式

規則表示式(Regular expression)最早是由數學家 Stephen Kleene 於 1956 年提出，主要用於字元字串格式比對，後來在資訊領域廣為應用。Python 提供一些支援規則表示式操作的標準 API，以下將從如何定義規則表示式開始介紹。

11.3.1　簡介規則表示式

如果有個字串，想根據某個字元或字串切割，可以使用 str 的 split()方法，它會傳回切割後各子字串組成的 list。例如：

```
>>> 'Justin,Monica,Irene'.split(',')
['Justin', 'Monica', 'Irene']
>>> 'JustinOrzMonicaOrzIrene'.split('Orz')
['Justin', 'Monica', 'Irene']
>>> 'Justin\tMonica\tIrene'.split('\t')
['Justin', 'Monica', 'Irene']
>>>
```

如果切割字串的依據不單只是某個字元或子字串，而是任意單一數字呢？例如，想將'Justin1Monica2Irene'依數字切割呢？這個時候 str 的 split()派不上用場，你需要的是規則表示式。

在 Python 中，使用 re 模組來支援規則表示式。例如，若想切割字串，可以使用 re.split()函式：

```
>>> import re
>>> re.split(r'\d', 'Justin1Monica2Irene')
['Justin', 'Monica', 'Irene']
>>> re.split(r',', 'Justin,Monica,Irene')
['Justin', 'Monica', 'Irene']
>>> re.split(r'Orz', 'JustinOrzMonicaOrzIrene')
['Justin', 'Monica', 'Irene']
>>> re.split(r'\t', 'Justin\tMonica\tIrene')
['Justin', 'Monica', 'Irene']
>>>
```

在這邊使用了 re 模組的 split()函式，第一個參數要以字串來指定規則表示式，在規則表示式中，\d 表示符合一個數字。

實際上若使用 Python 的字串表示時，因為\在 Python 字串中被作為轉義（Escape）字元，因此要撰寫規則表示式時，必須撰寫為'\\d'，這樣當然很麻煩，幸而 Python 可以在字串前加上 r，表示這是個原始字串（Raw string），不要對\做任何轉義動作，因此**在撰寫規則表示式時，建議使用原始字串。**

Python 支援大多數標準的規則表示式，想使用 re 模組之前，認識規則表示式是必要的。

規則表示式基本上包括兩種字元：**字面字元（Literals）**與**詮譯字元（Metacharacters）**。**字面字元是指按照字面意義比對的字元**，像是方才在範例中指定的 Orz，指的是三個字面字元 O、r、z 的規則；**詮譯字元是不按照字面比對，在不同情境有不同意義的字元**。

例如^是詮譯字元，規則表示式^Orz 是指行首立即出現 Orz 的情況，也就是此時^表示一行的開頭，但規則表示式[^Orz]是指不包括 O 或 r 或 z 的比對，也就是在[]中時，^表示非之後幾個字元的情況。詮譯字元就像是程式語言中的控制結構之類的語法，**找出並理解詮譯字元想詮譯的概念，對於規則表示式的閱讀非常重要**。

字元表示

字母和數字在規則表示式中，都是按照字面意義比對，有些字元之前加上了\之後，會被當作詮譯字元，例如\t 代表按下 Tab 鍵的字元，下表列出 Python 對規則表示式支援的字元表示：

表 11.2 字元表示

字元	說明
字母或數字	比對字母或數字
\\	比對\字元
\num	8 進位數字 num（0 開頭或三個數字）表示字元碼點
\xhh	16 進位數字 hh 表示字元碼點
\uhhhh	16 進位數字 hhhh 表示字元碼點（Python 3.3）
\Uhhhhhhhh	16 進位數字 hhhhhhhh 表示字元碼點（Python 3.3）
\n	換行（\u000A）
\v	垂直定位（\u000B）
\f	換頁（\u000C）
\r	返回（\u000D）
\a	響鈴（\u0007）
\b	退格（\u0008）
\t	Tab（\u0009）

詮譯字元在規則表示式中有特殊意義，例如! $ ^ * ( ) + = { } [ ] | \ : . ? 等，若要比對這些字元，必須加上轉義（Escape）符號，例如要比對!，則必須使用\!，要比對$字元，則必須使用\$。**如果不確定哪些標點符號字元要加上轉義符號，可以在每個標點符號前加上\，例如比對逗號也可以寫\,**。

如果規則表示式為 XY，那麼表示比對「X 之後要跟隨著 Y」，如果想表示「X 或 Y」，可以使用 X|Y，如果有多個字元要以「或」的方式表示，例如「X 或 Y 或 Z」，可以使用稍後會介紹的字元類表示為[XYZ]。

字元類

規則表示式中，多個字元可以歸類在一起，成為一個**字元類（Character class）**，字元類會比對文字中是否有「任一個」字元符合字元類中某個字元。**規則表示式中被放在[]中的字元就成為一個字元類**。例如，若字串為'Justin1Monica2Irene3Bush'，想依 1 或 2 或 3 切割字串，規則表示式可撰寫為[123]：

```
>>> re.split(r'[123]', 'Justin1Monica2Irene3Bush')
['Justin', 'Monica', 'Irene', 'Bush']
>>>
```

規則表示式 123 連續出現字元 1、2、3，然而[]中的字元是「或」的概念，也就是[123]表示「1 或 2 或 3」，**|在字元類別只是個普通字元，不會被當作「或」來表示。**

字元類中可以使用連字號-作為字元類詮譯字元，表示一段文字範圍，例如要比對文字中是否有 1 到 5 任一數字出現，規則表示式為[1-5]，要比對文字中是否有 a 到 z 任一字母出現，規則表示式為[a-z]，要比對文字中是否有 1 到 5、a 到 z、M 到 W 任一字元出現，規則表示式可以寫為[1-5a-zM-W]。**字元類中可以使用^作為字元類詮譯字元，[^]則為反字元類（Negated character class）**，例如[^abc]會比對 a、b、c 以外的字元。

以下為字元類範例列表：

表 11.3 字元類

字元類	說明
[abc]	a 或 b 或 c 任一字元
[^abc]	a、b、c 以外的任一字元
[a-zA-Z]	a 到 z 或 A 到 Z 任一字元
[a-d[m-p]]	a 到 d 或 m 到 p 任一字元（聯集），等於[a-dm-p]
[a-z&&[def]]	a 到 z 且是 d、e、f 的任一字元（交集），等於[def]
[a-z&&[^bc]]	a 到 z 且不是 b 或 c 的任一字元(減集)，等於[ad-z]
[a-z&&[^m-p]]	a 到 z 且不是 m 到 p 的任一字元，等於[a-lq-z]

可以看到，字元類中可以再有字元類，把規則表示式想成是語言的話，字元類就像是其中獨立的子語言。

有些字元類很常用，例如經常會比對是否為 0 到 9 的數字，可以撰寫為[0-9]，或是撰寫為\d，這類字元被稱為**字元類縮寫**或**預定義字元類（Predefined character class）**，它們不用被包括在[]之中，下表列出可用的預定義字元類：

表 11.4 預定義字元類（預設支援 Unicode 模式）

預定義字元類	說明
.	任一字元
\d	比對任一數字字元
\D	比對任一非數字字元
\s	比對任一空白字元
\S	比對任一非空白字元，即[^\s]
\w	比對任一字元
\W	比對任一非字元，即[^\w]

就規則表示式本身的發展歷史來說，早期並不支援 Unicode，例如預定義字元類早期就未考量 Unicode 規範，像是\w 預設只比對 ASCII 字元，在其他程式語言中，可能必須藉由 API 設定為支援 Unicode 模式，才能比對 ASCII 字元以外的字元。

Python 3 預設就支援 Unicode 模式，預定義字元類在比對上不限於 ASCII 字元，例如\w 就可以比對中文字元：

```
>>> re.search(r'\w', '林')
<re.Match object; span=(0, 1), match='林'>
>>>
```

re 模組的 search()函式，可搜尋指定字串中是否有符合規則表示式的文字，若有會傳回 re.Match 物件，若無傳回 None，以上的範例可看出\w 可以比對到中文字元。

類似地，\d 在 Python 中，預設並不只比對 ASCII 數字 0 至 9，**1 2 3** 4 5 6 7 8 9 **0 1 2 3** 4 5 6 也可以比對成功，如果你撰寫.py 內容如下：

regex_unicode_numbers.py

```
import re

matched = re.findall(r'\d', '1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 ');
print(matched == list('1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 '))
```

re 模組的 findall()函式，會將符合規則表示式的文字收集至 list，因此執行結果會顯示 True。

如果想令預定義字元類僅支援 ASCII，例如令\d 等同於[0-9]，\s 等同於[\t\n\x0B\f\r]，\w 等同於[a-zA-Z0-9_]，必須設置旗標（flag），設置的方式將在 11.3.2 說明。

貪婪、逐步量詞

如果想判斷使用者輸入的手機號碼格式是否為 XXXX-XXXXXXX，其中 X 為數字，雖然規則表示式可以使用\d\d\d\d-\d\d\d\d\d\d\d，不過更簡單的寫法是\d{4}-\d{6}，{n}是**貪婪量詞（Greedy quantifier）**表示法的一種，表示前面的項目出現 n 次。下表列出可用的貪婪量詞：

表 11.5 貪婪量詞

貪婪量詞	說明
X?	X 項目出現一次或沒有
X*	X 項目出現零次或多次
X+	X 項目出現一次或多次
X{n}	X 項目出現 n 次
X{n,}	X 項目至少出現 n 次
X{n,m}	X 項目出現 n 次但不超過 m 次

貪婪量詞之所以貪婪，是因為看到貪婪量詞時，比對器（Matcher）會把符合量詞的文字全部吃掉，再逐步吐出（back-off）文字，看看是否符合貪婪量詞後的規則表示式，如果吐出的部份也符合就比對成功，結果就是**貪婪量詞會儘可能地找出長度最長的符合文字**。

例如文字 xfooxxxxxxfoo，若使用規則表示式.*foo 比對，比較器根據.*吃掉了整個 xfooxxxxxxfoo，之後吐出 foo 符合 foo 部份，得到的符合字串就是整個 xfooxxxxxxfoo。

若在貪婪量詞表示法後加上?，會成為**逐步量詞（Reluctant quantifier）**，又常稱為**懶惰量詞**，或**非貪婪（non-greedy）量詞**（相對於貪婪量詞來說），比對器是一邊吃，一邊比對文字是否符合量詞與之後的規則表示式，結果就是**逐步量詞會儘可能地找出長度最短的符合文字。**

例如文字 xfooxxxxxxfoo 若用規則表示式.*?foo 比對，比對器在吃掉 xfoo 後發現符合.*?foo，接著繼續吃掉 xxxxxxfoo 發現符合，所以得到 xfoo 與 xxxxxxfoo 兩個符合文字。

可以使用 re 模組的 findall()來實際看看兩個量詞的差別：

```
>>> re.findall(r'.*foo', 'xfooxxxxxxfoo')
['xfooxxxxxxfoo']
>>> re.findall(r'.*?foo', 'xfooxxxxxxfoo')
['xfoo', 'xxxxxxfoo']
>>>
```

以下是幾個 glob()可使用的比對範例：

- *.py 比對.py 結尾的字串。
- **/*test.py 跨目錄比對 test.py 結尾的路徑，在 glob()的 recursive 設定為 True 下，bookmark_test.py、command_test.py 都符合。
- ???符合三個字元，例如 123、abc 會符合。
- a?*.py 比對 a 之後至少一個字元，並以.py 結尾的字串。
- *[0-9]*比對的字串中要有一個數字。

glob()會以 list 傳回符合的路徑，iglob()的功能與 glob()相同，不過傳回迭代器。以下製作一個範例，可指定 glob()可接受的模式，搜尋指定路徑下符合的檔案：

files_dirs glob_search.py

```
import sys, os, glob

try:
    path = sys.argv[1]
    pattern = sys.argv[2]
except IndexError:
    print('請指定搜尋路徑與 glob 模式')
    print('例如：python glob_search.py c:\workspace **/*.py')
else:
    os.chdir(path)
    for p in glob.iglob(pattern, recursive = True):
        print(p)
```

一個執行結果如下：

```
C:\workspace\files_dirs>python glob_search.py c:\workspace **\*.pyc
logging_demo\__pycache__\logconf.cpython-37.pyc
```

11.5 URL 處理

Python 常用來撰寫 Web 爬蟲（Scraper），複雜的 Web 爬蟲程式，會使用專門的程式庫來處理，然而，對於一些簡單的 HTML 頁面資訊擷取，只需要一點點的 URL 及 HTTP 概念，結合目前所學，就可以使用 urllib 來輕鬆完成任務。

11.5.1　淺談 URL 與 HTTP

雖然對於一些簡單的場合，在不認識 URL 與 HTTP 細節的情況下，也可以使用 urllib 完成任務，然而知道些細節，可以讓你完成更多的操作，因此這邊先針對後續討論 urllib 時，提供一些夠用的基礎。

URL 規範

Web 應用程式的文件、檔案等資源是放在 Web 網站上，而 Web 網站棲身於廣大網路之中，必須要有個方式，告訴瀏覽器到哪裡取得文件、檔案等資源，通常會聽到有人這麼說：「要指定 URL」。

URL 中的 U，早期代表 Universal（萬用），標準化之後代表 Uniform（統一），因此目前 URL 全名為 Uniform Resource Locator。正如名稱指出，URL 主要目的，是以文字方式說明網路上的資源如何取得，就早期的〈RFC 1738〉[11]規範來看，URL 的主要語法格式為：

```
<scheme>:<scheme-specific-part>
```

協議（scheme）指定了以何種方式取得資源，一些協議名稱的例子有：

- ftp（檔案傳輸協定，File Transfer protocol）
- http（超文件傳輸協定，Hypertext Transfer Protocol）
- mailto（電子郵件）
- file（特定主機檔案名稱）

提示 »» urllib 實際上也能處理 FTP 等協定，然而作為一個入門介紹，後續會將焦點放在 HTTP。

協議之後跟隨冒號，協議特定部份（scheme-specific-part）的格式依協議而定，通常會是：

```
//<使用者>:<密碼>@<主機>:<埠號>/<路徑>
```

[11] RFC 1738：tools.ietf.org/html/rfc1738

舉例來說，若主機名稱為 openhome.cc，要以 HTTP 協定取得 Gossip 資料夾中 index.html 文件，連接埠號 8080，必須使用以下的 URL：

```
http://openhome.cc:8080/Gossip/index.html
```

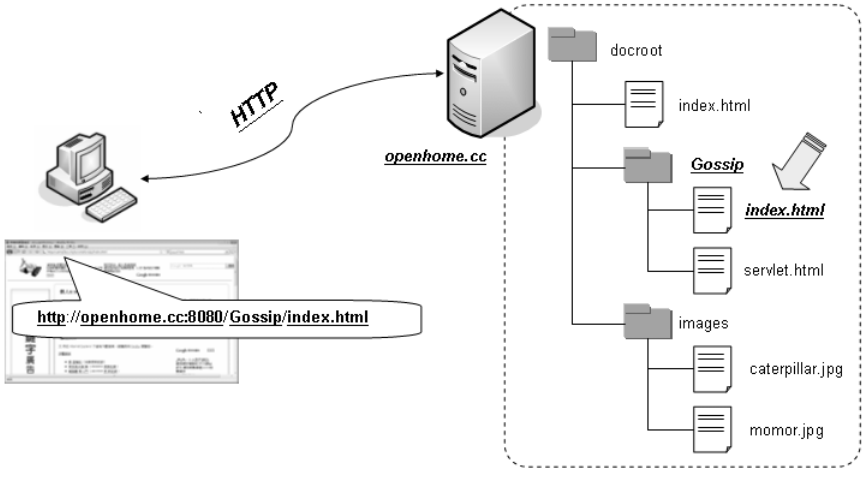

圖 11.2 以 URL 指定資源位置等資訊

提示 »» 由於一些歷史性的原因，URL 後來成為 URI 規範的子集，有興趣可以參考維基百科的〈Uniform Resource Identifier〉[12]條目；為了符合 `urllib` 名稱，底下仍舊使用 URL 此名稱來進行說明。

目前來說，請求 Web 應用程式主要是透過 HTTP 通訊協定，通訊協定是電腦間對談溝通的方式，例如客戶端要跟伺服器要求連線，假設就是跟伺服器說聲 `CONNECT`，伺服器回應 `PASSWORD` 表示要求密碼，客戶端進一步跟伺服器說聲 `PASSWORD 1234`，表示這是所需的密碼，諸如此類。

12 Uniform Resource Identifier：en.wikipedia.org/wiki/Uniform_Resource_Identifier

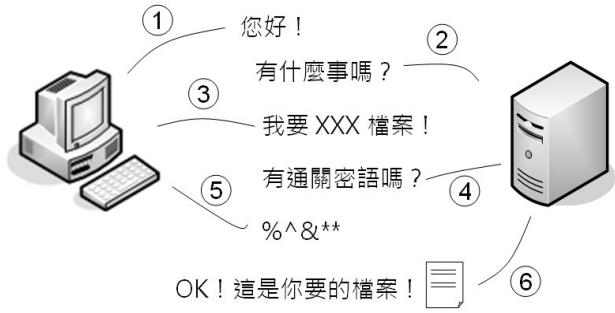

圖 11.3 通訊協定是電腦間溝通的一種方式

瀏覽器跟 Web 網站間使用的溝通方式基本上是 HTTP，HTTP 定義了 GET、POST、PUT、DELETE、HEAD、OPTIONS、TRACE 等請求方式，就使用 urllib 來說，最常運用 GET 與 POST，底下就針對 GET 與 POST 來進行說明。

GET 請求

GET 請求顧名思義，就是向 Web 網站取得指定的資源，在發出 GET 請求時，必須告訴 Web 網站請求資源的 URL，以及一些標頭（Header）資訊，例如一個 GET 請求的發送範例如下所示：

圖 11.4 GET 請求範例

上圖請求標頭提供了 Web 網站一些瀏覽器相關的資訊，Web 網站可以使用這些資訊來進行回應處理。例如 Web 網站可從 User-Agent 得知使用者的瀏覽器種類與版本，從 Accept-Language 了解瀏覽器接受哪些語系的內容回應等。

請求參數通常是使用者發送給 Web 網站的資訊，Web 網站有了這些資訊，可以進一步針對使用者請求進行正確的回應，請求參數是路徑之後跟隨問號

（?），然後是請求參數名稱與請求參數值，中間以等號（=）表示成對關係。若有多個請求參數，以&字元連接。使用 GET 的方式發送請求，瀏覽器的網址列上也會出現請求參數資訊。

圖 11.5 GET 請求參數會出現在網址列

GET 請求可以發送的請求參數長度有限，這依瀏覽器而有所不同，Web 網站也會設定長度上的限制，大量的資料不適合用 GET 方法來進行請求，Web 應用程式可改為接受 POST 請求。

POST 請求

對於大量或複雜的資訊發送（例如檔案上傳），可採用 POST 來進行發送，一個 POST 發送的範例如下所示：

圖 11.6 POST 請求範例

POST 將請求參數移至最後的訊息本體（Message body），由於訊息本體的內容長度不受限制，大量資料的發送可使用 POST 方法，由於請求參數移至訊息本體，網址列上也就不會出現請求參數，對於較敏感的資訊，像是密碼，即使長度不長，通常也會改用 POST 的方式發送，以避免因為出現在網址列上而被直接窺看。

注意»»» 雖然在 POST 請求時，請求參數不會出現在網址列上，然而在非加密連線的情況下，若請求被第三方擷取了，請求參數仍然是一目瞭然。