
Argparse 教學

發 3.14.0a1

Guido van Rossum and the Python development team

11 月 11, 2024

Python Software Foundation
Email: docs@python.org

Contents

1	概念	2
2	基本用法	2
3	位置引數的介紹	3
4	可選引數的介紹	4
4.1	短選項	5
5	組合位置引數和可選引數	6
6	更進階一點	9
6.1	指定不明確的引數	10
6.2	相互衝突的選項	11
7	如何翻譯 argparse 輸出	12
8	Custom type converters	13
9	結論	13

作者

Tshepang Mbambo

本教學旨在簡要介紹 argparse 這個 Python 標準函式庫中推薦的命令列剖析模組。

備註

另外還有兩個模組可以完成相同的任務，即 `getopt`（相當於 C 語言中的 `getopt()`）和已用的 `optparse`。另請注意，`argparse` 是基於 `optparse` 的，因此在用法上非常相似。

1 概念

讓我們透過使用 **ls** 指令來展示我們將在本介紹教學中探索的功能類型：

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILES (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

我們可以從這四個命令中學到一些概念：

- **ls** 命令即便在沒有任何選項的情況下執行仍非常有用。它預設顯示目前目錄的內容。
- 如果我們想要看到比它預設提供更多的內容，我們也需要多告訴它一點。在本例中，我們希望它顯示不同的目錄 `pypy`，我們做的是指定所謂的位置引數。之所以如此命名是因爲程式應該只根據該值在命令列中出現的位置來知道如何處理該值。這個概念與 **cp** 這樣的指令更相關，其最基本的用法是 `cp SRC DEST`。第一個是你想要複製的位置，第二個是你想要複製過去的位置。
- 現在假設我們想要改變程式的行爲。在我們的範例中，我們顯示每個檔案的更多資訊，而不僅是顯示檔案名稱。在這種情況下，`-l` 被稱爲可選引數。
- 這是幫助文字的片段。它非常有用，因爲當你遇到以前從未使用過的程式時，只需閱讀其幫助文字即可了解它的工作原理。

2 基本用法

讓我們從一個非常簡單的例子開始，它（幾乎）什麼都不做：

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

程式碼執行結果如下：

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

options:
  -h, --help  show this help message and exit
$ python prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

這是發生的事情：

- 執行不帶任何選項的程式本不會在標準輸出中顯示任何內容。不太有用。

- 第二個開始能顯現 `argparse` 模組的有用之處。我們幾乎什麼也做，但我們已經收到了一個很好的幫助訊息。
- `--help` 選項也可以縮寫成 `-h`，是我們能隨意獲得的唯一選項（即無需指定它）。指定任何其他內容都會導致錯誤。但即便如此，我們也還是輕鬆地獲得了有用的使用資訊。

3 位置引數的介紹

例如：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print(args.echo)
```

執行這段程式碼：

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

options:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

這是會發生的事情：

- 我們新增了 `add_argument()` 方法，我們用它來指定程式願意接受哪些命令列選項。在本例中，我將其命名為 `echo`，以便與其功能一致。
- 現在呼叫我們的程式時需要指定一個選項。
- `parse_args()` 方法實際上從指定的選項中回傳一些資料，在本例中為 `echo`。
- 該變數是某種形式的「魔法」，`argparse` 可以自由執行（即無需指定該值儲存在哪個變數中）。你還會注意到，它的名稱與提供給方法 `echo` 的字串引數相符。

但請注意，儘管幫助顯示看起來不錯，但它目前還沒有發揮出應有的用處。例如，我們看到 `echo` 作為位置引數，但除了猜測或閱讀原始程式碼之外，我們不知道它的作用。那麼，我們來讓它變得更有用一點：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print(args.echo)
```

然後我們得到：

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo          echo the string you use here

options:
  -h, --help  show this help message and exit
```

現在來做一些更有用處的事情：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print(args.square**2)
```

程式碼執行結果如下：

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

進展不太順利。這是因為，除非我們另有說明，argparse 會將我們給它的選項視作字串。因此，讓我們告訴 argparse 將該輸入視作整數：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print(args.square**2)
```

程式碼執行結果如下：

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

順利進展。現在該程式甚至可以在繼續操作之前因錯誤的非法輸入而退出。

4 可選引數的介紹

到目前為止，我們一直在討論位置引數。我們來看看如何新增可選引數：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

接者是結果：

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

options:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

這是發生的事情：

- 程式被編寫在指定 `--verbosity` 時顯示一些內容，在未指定時不顯示任何內容。
- 為了表示該選項實際上是可選的，使用它來執行程式不會出現錯誤。請注意，預設情況下，如果未使用可選引數，則相關變數（在本例中 `args.verbosity`）將被賦予 `None` 作值，這就是它未能通過 `if` 陳述式真值測試的原因。
- 幫助訊息有點不同。
- 當使用 `--verbosity` 選項時必須要指定一些值，任何值都可以。

在上面的例子中，`--verbosity` 接受任意的整數，但對我們的程式來只接受兩個輸入值，`True` 或 `False`。所以我們來修改一下程式碼使其符合：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

接著是結果：

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

options:
  -h, --help  show this help message and exit
  --verbose  increase output verbosity
```

這是發生的事情：

- 這個選項現在更像是一個旗標，而不是需要值的東西。我們甚至更改了選項的名稱以符合這個想法。請注意，我們現在指定一個新的關鍵字 `action`，其指定值 `"store_true"`。這意味著，如果指定了該選項，則將值 `True` 指派給 `args.verbose`。不指定它代表 `False`。
- 當你指定一個值時，它會本著旗標的實際精神來抱怨。
- 請注意不同的幫助文字。

4.1 短選項

如果你熟悉命令列用法，你會注意到我尚未提及選項的簡短版本。這很簡單：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

而這：

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
```

(繼續下一頁)

```
usage: prog.py [-h] [-v]

options:
  -h, --help      show this help message and exit
  -v, --verbose   increase output verbosity
```

請注意，新功能也反映在幫助文字中。

5 組合位置引數和可選引數

我們的程式的複雜性不斷增加：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print(f"the square of {args.square} equals {answer}")
else:
    print(answer)
```

然後現在的輸出結果：

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- 我們帶回了位置引數，因而被抱怨。
- 請注意，順序並不重要。

我們讓這個程式擁有多個訊息詳細級別 (verbosity) 之值的能力，實際使用它們：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

接者是結果：

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

除了最後一個外都看起來正常，它透露了我們程式中的一個錯誤。我們可透過限制 `--verbosity` 選項可以接受的值來修復它：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

接者是結果：

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity {0,1,2}
                        increase output verbosity
```

請注意，更改也會反映在錯誤訊息和幫助字串中。

現在，讓我們使用另一種常見方法來玩玩訊息詳細級^[4]。它也與 CPython 執行檔處理其自身訊息詳細級^[5]引數的方式相符（請見 `python --help` 的輸出）：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity == 1:
```

(繼續下一頁)

```

    print(f"{args.square}^2 == {answer}")
else:
    print(answer)

```

我們已經介紹過另一個操作“count”用來計算指定的選項出現的次數。

```

$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square                display a square of a given number

options:
  -h, --help            show this help message and exit
  -v, --verbosity       increase output verbosity
$ python prog.py 4 -vvv
16

```

- 是的，現在它更像是我們上一版📖本中的旗標（類似於 `action="store_true"`），這應該可以解釋抱怨的原因。
- 它的行📖也類似“store_true”操作。
- 現在這📖示範了“count”動作的作用。你可能以前見過這種用法。
- 如果你不指定 `-v` 旗標，則該旗標被視📖具有 `None` 值。
- 正如預期的那樣，指定長形式旗標，我們應該得到相同的輸出。
- 遺憾的是，我們的幫助輸出對於我們📖本獲得的新功能📖📖有提供太多資訊，但我們都可以透過改進📖本的文件來解📖這個問題（例如：透過 `help` 關鍵字引數）。
- 最後的輸出透露了我們程式中的一個錯誤。

讓我們來解📖問題：

```

import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)

```

這就是它給出的：

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: '>=' not supported between instances of 'NoneType' and 'int'
```

- 第一次輸出順利進行，[☞](#)修復了我們之前遇到的錯誤。也就是[☞](#)，我們希望任何 ≥ 2 的值都盡可能詳細。
- 第三個輸出不太好。

我們來修復這個錯誤：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

我們剛剛引入了另一個關鍵字 `default`。我們將其設為 0，以便使其與其他 `int` 值進行比較。請記住，預設情況下，如果未指定可選引數，它將獲得 `None` 值，[☞](#)且不能與 `int` 值進行比較（因此會出現 `TypeError` 例外）。

而且：

```
$ python prog.py 4
16
```

僅憑我們迄今為止所學到的知識就可以做到很多事情了，不過其實這樣只有學到一點皮毛而已。`argparse` 模組非常強大，在結束本教學之前我們會對它進行更多探索。

6 更進階一點

如果我們想擴充我們的小程式來執行其他次方的運算，而不僅是平方：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"{args.x} to the power {args.y} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.x}^{args.y} == {answer}")
```

(繼續下一頁)

```
else:
    print(answer)
```

結果：

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                  the base
  y                  the exponent

options:
  -h, --help          show this help message and exit
  -v, --verbosity

$ python prog.py 4 2 -v
4^2 == 16
```

請注意，到目前為止，我們一直在使用詳細級別來更改顯示的文字。以下範例使用詳細級別來顯示更多文字：

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print(f"Running '{__file__}'")
if args.verbosity >= 1:
    print(f"{args.x}^{args.y} == ", end="")
print(answer)
```

結果：

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

6.1 指定不明確的引數

當指定一個引數是位置引數還是引數會有歧義，可以使用 `--` 來告訴 `parse_args()` 之後的所有內容都是位置引數：

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('-n', nargs='+')
>>> parser.add_argument('args', nargs='*')

>>> # ambiguous, so parse_args assumes it's an option
>>> parser.parse_args(['-f'])
usage: PROG [-h] [-n N [N ...]] [args ...]
PROG: error: unrecognized arguments: -f
```

```
>>> parser.parse_args(['--', '-f'])
Namespace(args=['-f'], n=None)

>>> # ambiguous, so the -n option greedily accepts arguments
>>> parser.parse_args(['-n', '1', '2', '3'])
Namespace(args=[], n=['1', '2', '3'])

>>> parser.parse_args(['-n', '1', '--', '2', '3'])
Namespace(args=['2', '3'], n=['1'])
```

6.2 相互冲突的選項

到目前為止，我們一直在使用 `argparse.ArgumentParser` 實例的兩種方法。讓我們介紹第三個，`add_mutually_exclusive_group()`，它允許我們指定彼此衝突的選項。我們還可以更改程式的其餘部分，以使得新功能更有意義：我們將引入 `--quiet` 選項，該選項與 `--verbose` 選項相反：

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

我們的程式現在更簡單了，我們因功能展示失去了一些功能，但無論如何，以下這是輸出：

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

這應該很容易理解。我新增了最後一個輸出，以便看到所獲得的靈活性，即可以混合長形式與短形式選項。

在我們結束之前，你可能想告訴使用者你的程式的主要目的，以防他們不知道：

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
```

(繼續上一頁)

```
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print(f"{args.x} to the power {args.y} equals {answer}")
else:
    print(f"{args.x}^{args.y} == {answer}")
```

請注意用法文字中的細微差別。注意 `[-v | -q]`，它告訴我們可以使用 `-v` 或 `-q`，但不能同時使用：

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x                the base
  y                the exponent

options:
  -h, --help      show this help message and exit
  -v, --verbose
  -q, --quiet
```

7 如何翻譯 argparse 輸出

argparse 模組的輸出，例如幫助文字和錯誤訊息，都可以透過使用 `gettext` 模組進行翻譯。這允許應用程式能輕鬆本地化 argparse 生成的訊息。另請參閱 [i18n-howto](#)。

例如，在此 argparse 輸出中：

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x                the base
  y                the exponent

options:
  -h, --help      show this help message and exit
  -v, --verbose
  -q, --quiet
```

字串 `usage:`、`positional arguments:`、`options:` 和 `show this help message and exit` 都是可被翻譯的。

為了翻譯這些字串，必須先將它們提取到 `.po` 檔案中。例如，使用 [Babel](#) 執行下列命令：

```
$ pybabel extract -o messages.po /usr/lib/python3.12/argparse.py
```

此命令將從 `argparse` 模組中提取出所有可翻譯的字串，並將它們輸出到名為 `messages.po` 的檔案中。這個指令假設你的 `Python` 是安裝在 `/usr/lib` 中。

你可以使用以下方法找到 `argparse` 模組在系統上的位置：

```
import argparse
print (argparse.__file__)
```

一旦翻譯了 .po 檔案中的訊息，使用 gettext 安裝了翻譯，argparse 將能顯示翻譯後的訊息。若要在 argparse 輸出中翻譯你自己的字串，請使用 gettext。

8 Custom type converters

The `argparse` module allows you to specify custom type converters for your command-line arguments. This allows you to modify user input before it's stored in the `argparse.Namespace`. This can be useful when you need to pre-process the input before it is used in your program.

When using a custom type converter, you can use any callable that takes a single string argument (the argument value) and returns the converted value. However, if you need to handle more complex scenarios, you can use a custom action class with the **action** parameter instead.

For example, let's say you want to handle arguments with different prefixes and process them accordingly:

```
import argparse

parser = argparse.ArgumentParser(prefix_chars='-+')

parser.add_argument('-a', metavar='<value>', action='append',
                    type=lambda x: ('-', x))
parser.add_argument('+a', metavar='<value>', action='append',
                    type=lambda x: ('+', x))

args = parser.parse_args()
print (args)
```

結果：

```
$ python prog.py -a value1 +a value2
Namespace(a=[('-', 'value1'), ('+', 'value2')])
```

在這個範例當中，我們：

- Created a parser with custom prefix characters using the `prefix_chars` parameter.
- Defined two arguments, `-a` and `+a`, which used the `type` parameter to create custom type converters to store the value in a tuple with the prefix.

Without the custom type converters, the arguments would have treated the `-a` and `+a` as the same argument, which would have been undesirable. By using custom type converters, we were able to differentiate between the two arguments.

9 結論

`argparse` 模組提供的功能比此篇內容的要多得多。它的文件非常詳細與透徹，有很多範例。讀完本教學後，你應該可以輕鬆消化它們，而不會感到不知所措。