

---

# Argparse 자습서

릴리스 3.13.7

Guido van Rossum and the Python development team

9월 01, 2025

## Contents

1	개념	2
2	기본	2
3	위치 인자 소개	3
4	옵션 인자 소개	4
4.1	짧은 옵션	6
5	위치 및 옵션 인자 결합하기	6
6	조금 더 발전시키기	10
6.1	Specifying ambiguous arguments	11
6.2	충돌하는 옵션들	11
7	How to translate the argparse output	13
8	Custom type converters	14
9	맺음말	14

---

저자

Tshepang Mbambo

이 자습서는 파이썬 표준 라이브러리에서 권장하는 명령행 파싱 모듈인 `argparse`에 대한 소개입니다.

### 참고

The standard library includes two other libraries directly related to command-line parameter processing: the lower level `optparse` module (which may require more code to configure for a given application, but also allows an application to request behaviors that `argparse` doesn't support), and the very low level `getopt` (which specifically serves as an equivalent to the `getopt()` family of functions available to C programmers). While neither of those modules is covered directly in this guide, many of the core concepts in `argparse` first originated in `optparse`, so some aspects of this tutorial will also be relevant to `optparse` users.

# 1 개념

**ls** 명령을 사용하여 이 입문서에서 다룰 기능들을 살펴봅시다:

```
$ 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.
...
```

네 가지 명령에서 배울 수 있는 몇 가지 개념들입니다:

- **ls** 명령은 옵션 없이 실행될 때도 유용합니다. 기본적으로 현재 디렉터리의 내용을 표시합니다.
- 기본적으로 제공하는 것 이상으로 원한다면, 조금 더 말합니다. 이 경우에는 다른 디렉터리인 `pypy` 를 표시하기를 원합니다. 우리가 한 것은 위치 인자라고 알려진 것을 지정하는 것입니다. 프로그램이 명령행에 표시되는 위치를 기준으로 값을 어떻게 처리해야 하는지를 알아야 하므로 이런 이름이 사용됩니다. 이 개념은 **cp** 와 같은 명령에 더 적절합니다. 가장 기본적인 사용법은 `cp SRC DEST` 입니다. 첫 번째 위치는 복사하고자 하는 것 이고 두 번째 위치는 사본을 저장할 곳 입니다.
- 자, 프로그램의 행동을 바꾸고 싶다고 합시다. 이 예에서는 파일 이름만 표시하는 대신 각 파일에 대한 정보를 더 많이 표시합니다. 이 경우 `-l` 은 옵션 인자로 알려져 있습니다.
- 이것이 도움말 텍스트입니다. 이전에 사용해보지 않은 프로그램을 접했을 때 도움말 텍스트를 읽는 것만으로 작동하는 방식을 이해할 수 있다는 점에서 매우 유용합니다.

# 2 기본

(거의) 아무것도 하지 않는 아주 간단한 예제로 시작합니다:

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

다음은 코드를 실행한 결과입니다:

```
$ 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
```

일어난 일은 이렇습니다:

- 옵션 없이 스크립트를 실행하면 아무것도 표준 출력에 표시되지 않습니다. 별로 유용하지 않습니다.
- 두 번째는 `argparse` 모듈의 쓸모를 보여주기 시작합니다. 거의 아무것도 하지 않았지만 이미 도움 말을 얻었습니다.
- `--help` 옵션은, `-h` 로 단축할 수도 있습니다, 무료로 얻을 수 있는 유일한 옵션입니다 (즉, 지정할 필요가 없습니다). 다른 값을 지정하면 오류가 발생합니다. 그러나 그때조차도 우리는 사용 안내를 얻습니다, 여전히 공짜입니다.

### 3 위치 인자 소개

예:

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

코드를 실행합니다:

```
$ 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
```

이런 일이 일어났습니다:

- `add_argument()` 메서드를 추가했습니다. 이 메서드는 프로그램이 받고 싶은 명령행 옵션을 지정하기 위해 사용합니다. 이 경우 기능과 일치하도록 `echo` 라고 이름 붙였습니다.
- 이제 프로그램을 호출하려면 옵션을 지정해야 합니다.
- `parse_args()` 메서드는 실제로 지정된 옵션으로부터 온 데이터를 돌려줍니다. 이 경우에는 `echo` 입니다.
- 변수는 `argparse` 이 공짜로 수행하는 일종의 ‘마법’ 입니다 (즉, 값이 저장되는 변수를 지정할 필요가 없습니다). 또한, 그 이름이 메서드에 주어진 문자열 인자 `echo` 와 일치함을 알 수 있습니다.

그러나 도움말이 멋지게 보이지만, 현재로서는 가능한 최선이 아닙니다. 예를 들어 `echo` 가 위치 인자임을 볼 수 있지만, 추측하거나 소스 코드를 읽는 것 외에는 그것이 무엇을 하는지 모릅니다. 그럼 좀 더 유용하게 만들어 봅시다:

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

그러면 이렇게 됩니다:

```
$ 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
```

이제, 뭔가 더 쓸모있는 일을 하는 것은 어떻습니까:

```
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)
```

다음은 코드를 실행한 결과입니다:

```
$ 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'
```

잘 안됐습니다. 우리가 달리 지시하지 않는다면, argparse 는 우리가 준 옵션들을 문자열로 취급하기 때문입니다. 그럼, argparse 에게 그 입력을 정수로 취급하라고 알려줍시다:

```
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)
```

다음은 코드를 실행한 결과입니다:

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

잘 됩니다. 이제 이 프로그램은 잘못된 입력이 올 때 더 진행하지 않고 종료하기조차 합니다.

## 4 옵션 인자 소개

지금까지 우리는 위치 인자를 다뤘습니다. 옵션 인자를 추가하는 방법에 대해 살펴봅시다:

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

출력은 이렇습니다:

```
$ 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
```

일어난 일은 이렇습니다:

- 이 프로그램은 `--verbosity` 가 지정되었을 때 어떤 것을 표시하고 그렇지 않을 때는 아무것도 표시하지 않도록 작성되었습니다.
- 옵션이 실제로 선택 사항임을 확인하기 위해, 이 옵션을 사용하지 않고 프로그램을 실행할 때 오류가 없습니다. 기본적으로 옵션 인자가 사용되지 않는다면 관련 변수(이 경우 `args.verbosity`)는 값으로 `None` 이 주어집니다. 이 때문에 `if` 문의 논리값 검사가 실패합니다.
- 도움말 메시지가 약간 달라졌습니다.
- `--verbosity` 옵션을 사용할 때, 어떤 값을 지정해야 합니다. 어떤 값이건 상관없습니다.

위의 예제는 `--verbosity` 에 임의의 정숫값을 허용하지만, 우리의 간단한 프로그램에서는 실제로 `True` 또는 `False` 두 값만 쓸모 있습니다. 그것에 맞게 코드를 수정합시다:

```
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")
```

출력은 이렇습니다:

```
$ 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
```

일어난 일은 이렇습니다:

- 이 옵션은 이제 값을 요구하는 것이 아니라 플래그입니다. 이 개념과 일치하도록 옵션의 이름을 변경하기까지 했습니다. 새로운 키워드 `action` 을 지정하고, `"store_true"` 값을 지정했습니다. 이것은, 옵션이 지정되면 `args.verbose` 에 값 `True` 를 대입하라는 뜻입니다. 지정하지 않으면 묵시적으로 `False` 입니다.
- 값을 지정하면 불평하는데, 플래그의 정의를 따르고 있습니다.
- 도움말 텍스트가 바뀐 것을 확인하십시오.

## 4.1 짧은 옵션

명령행 사용법에 익숙하다면 짧은 옵션 버전에 관한 내용을 아직 다루지 않았음을 알 수 있을 겁니다. 아주 간단합니다:

```
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")
```

그러면 이렇게 됩니다:

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

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

새로운 기능은 도움말 텍스트에도 반영됩니다.

## 5 위치 및 옵션 인자 결합하기

프로그램이 점점 복잡해지고 있습니다:

```
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)
```

이제 출력은 이렇게 됩니다:

```
$ 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
```

- 위치 인자를 다시 도입했기 때문에, 불평합니다.
- 순서는 중요하지 않습니다.

이 프로그램에 여러 상세도를 지정할 수 있도록 하는 능력을 다시 부여하고, 실제로 그것을 사용하는 것은 어떨까요:

```
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)
```

출력은 이렇습니다:

```
$ 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
```

우리 프로그램의 버그를 드러내는 마지막 것을 제외하고는 그럴듯해 보입니다. --verbosity 옵션이 받아들일 수 있는 값을 제한해서 고쳐봅시다:

```
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)
```

출력은 이렇습니다:

```
$ 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
```

(다음 페이지에 계속)

```
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
```

변경 내용은 오류 메시지와 도움말 문자열에도 반영됩니다.

이제 상세도를 다루는 다른 접근법을 사용해 봅시다, 이 방법은 꽤 널리 사용됩니다. 또한, 이 방법은 CPython 실행 파일이 자신의 상세도를 처리하는 방식과도 일치합니다 (python --help 의 결과를 확인하십시오):

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

특정 옵션을 지정한 횟수를 계산하기 위해 “count” 라는 또 다른 액션을 도입했습니다.

```
$ 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
```

- 예, 이제 이전 버전의 스크립트처럼 (action="store\_true" 와 유사하게) 플래그가 되었습니다. 출력되는 불평이 설명됩니다.
- 또한 “store\_true” 액션과 비슷하게 작동하기도 합니다.



- 이제 여기에서 “count” 액션이 제공하는 것을 보여줍니다. 이런 종류의 사용법을 전에도 보았을 것입니다.
- 그리고, -v 플래그를 지정하지 않으면 그 플래그는 None 값으로 간주합니다.
- 예측하듯이, 플래그의 긴 형식을 지정하면, 같은 출력이 얻어져야 합니다.
- 안타깝게도 스크립트가 얻은 새로운 기능에 대한 도움말 출력은 그다지 유익하지 않지만, 스크립트의 문서를 개선하면 항상 해결할 수 있습니다 (예, help 키워드 인자를 사용해서).
- 마지막 출력은 우리 프로그램의 버그를 노출합니다.

고칩시다:

```
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)
```

그러면 이렇게 됩니다:

```
$ 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'
```

- 첫 번째 출력은 잘 동작하고, 앞에서 나온 버그를 고칩니다. 즉, 2보다 크거나 같은 (>=) 모든 값을 최대의 상세도로 취급하고 싶습니다.
- 세 번째 결과가 좋지 않습니다.

이 버그를 고쳐 봅시다:

```
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:
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

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

또 다른 키워드 `default` 를 소개했습니다. 다른 `int` 값과 비교하기 위해 0 으로 설정했습니다. 기본적으로, 옵션 인자가 지정되지 않으면 `None` 값을 갖게 되고, 그것은 `int` 값과 비교될 수 없음을 (그래서 `TypeError` 예외를 일으킵니다) 기억하십시오.

그리고:

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

여러분은 지금까지 배운 것만으로도 아주 멀리 갈 수 있으며, 우리는 단지 표면을 긁었을 뿐입니다. `argparse` 모듈은 매우 강력합니다. 이 자습서를 끝내기 전에 좀 더 탐색해 보겠습니다.

## 6 조금 더 발전시키기

우리의 작은 프로그램을 확장하여 제공만이 아닌 다른 거듭제곱을 수행하기를 원하면 어떻게 될까요:

```
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}")
else:
    print(answer)
```

출력:

```
$ 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     show verbosity count

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

지금까지는 표시되는 텍스트를 변경 하기 위해 상세도를 사용했습니다. 다음 예제는 대신 더 많은 텍스트를 표시하기 위해 상세도를 사용합니다:

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

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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)
```

출력:

```
$ 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 Specifying ambiguous arguments

When there is ambiguity in deciding whether an argument is positional or for an argument, `--` can be used to tell `parse_args()` that everything after that is a positional argument:

```
>>> 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 충돌하는 옵션들

지금까지 우리는 `argparse.ArgumentParser` 인스턴스의 두 가지 메서드로 작업 해왔습니다. 세 번째를 소개합니다, `add_mutually_exclusive_group()`. 이것은 서로 배타적인 옵션을 지정할 수 있도록 합니다. 새로운 기능을 더 잘 이해할 수 있도록 프로그램의 나머지 부분을 변경해 보겠습니다: `--quiet` 옵션을 도입하는데 `--verbose`의 반대입니다:

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
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}")
```

프로그램은 이제 더 간단 해졌으며, 데모를 위해 일부 기능을 잃어버렸습니다. 어쨌든, 출력은 이렇습니다:

```
$ 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
```

따라가기 쉽습니다. 여러분이 얻는 유연성을 볼 수 있도록 마지막 출력을 추가했습니다. 즉, 긴 형식 옵션을 짧은 형식 옵션과 섞어 쓸 수 있습니다.

결론을 내리기 전에, 여러분은 아마도 사용자들이 모를 경우를 대비해서 프로그램의 주요 목적을 알려주기를 원할 것입니다:

```
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")
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}")
```

사용법 텍스트의 약간의 차이점에 유의하십시오. [-v | -q] 에 주목해야 하는데, -v 나 -q 를 사용할 수 있지만 동시에 둘 다를 사용할 수는 없다는 뜻입니다:

```
$ python prog.py --help
```

(다음 페이지에 계속)

```
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 How to translate the argparse output

The output of the `argparse` module such as its help text and error messages are all made translatable using the `gettext` module. This allows applications to easily localize messages produced by `argparse`. See also [i18n-howto](#).

For instance, in this `argparse` output:

```
$ 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
```

The strings `usage:`, `positional arguments:`, `options:` and `show this help message and exit` are all translatable.

In order to translate these strings, they must first be extracted into a `.po` file. For example, using [Babel](#), run this command:

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

This command will extract all translatable strings from the `argparse` module and output them into a file named `messages.po`. This command assumes that your Python installation is in `/usr/lib`.

You can find out the location of the `argparse` module on your system using this script:

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

Once the messages in the `.po` file are translated and the translations are installed using `gettext`, `argparse` will be able to display the translated messages.

To translate your own strings in the `argparse` output, use `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)
```

출력:

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

이 예에서,:

- 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 맺음말

`argparse` 모듈은 여기에 제시된 것보다 훨씬 많은 것을 제공합니다. 문서는 아주 상세하고 철저하며 예제가 풍부합니다. 이 자습서를 끝내면 압도감 없이 쉽게 소화할 수 있을 겁니다.