

---

# Tutorial de argparse

Versión 3.14.0rc2

Guido van Rossum and the Python development team

septiembre 01, 2025

Python Software Foundation  
Email: docs@python.org

## Índice general

1	Conceptos	2
2	Las bases	2
3	Introducción a los argumentos posicionales	3
4	Introducción a los argumentos opcionales	4
4.1	Opciones cortas	6
5	Combinar argumentos opcionales y posicionales	6
6	Un poco mas avanzado	10
6.1	Especificando argumentos ambiguos	11
6.2	Opciones conflictivas	12
7	Cómo traducir la salida de argparse	13
8	Custom type converters	14
9	Conclusión	14

---

### autor

Tshepang Mbambo

Este tutorial pretende ser una leve introducción a `argparse`, el módulo de análisis (*parsing*) de línea de comandos recomendado en la biblioteca estándar de Python.

### Nota

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 Conceptos

Vamos a mostrar el tipo de funcionalidad que vamos a explorar en este tutorial introductorio haciendo uso del comando **ls**:

```
$ 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.
...
```

Algunos conceptos que podemos aprender de los cuatro comandos:

- El comando **ls** es útil cuando se ejecuta sin ninguna opción en absoluto. Por defecto muestra el contenido del directorio actual.
- Si queremos hacer algo, mas allá de lo que provee por defecto, le contamos un poco mas. En este caso, queremos mostrar un directorio diferente, **pypy**. Lo que hicimos fue especificar lo que se conoce como argumento posicional. Se llama así porque el programa debe saber que hacer con ese valor, basado únicamente en función de donde aparece en la línea de comandos. Este concepto es mas relevante para un comando como **cp**, cuyo uso mas básico es **cp SRC DEST**. La primer posición es *lo que quieres copiar*, y la segunda posición es *a donde lo quieres copiar*.
- Ahora, digamos que queremos cambiar el comportamiento del programa. En nuestro ejemplo, mostramos mas información para cada archivo en lugar de solo mostrar los nombres de los archivos. El argumento **-l** en ese caso se conoce como argumento opcional.
- Este es un fragmento del texto de ayuda. Es muy útil porque puedes encontrar un programa que nunca has usado antes, y puedes darte cuenta de como funciona simplemente leyendo el texto de ayuda.

## 2 Las bases

Comencemos con un simple ejemplo, el cual no hace (casi) nada:

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

Lo siguiente es el resultado de ejecutar el código:

```
$ 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]
```

(continúe en la próxima página)

```
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Esto es lo que está pasando:

- Ejecutar el script sin ninguna opción da como resultado que no se muestra nada en *stdout*. No es tan útil.
- El segundo comienza a mostrar la utilidad del módulo `argparse`. No hemos hecho casi nada, pero ya recibimos un buen mensaje de ayuda.
- La opción `--help`, que también puede ser abreviada como `-h`, es la única opción que tenemos gratis (es decir, no necesitamos especificarla). Especificar cualquier otra cosa da como resultado un error. Pero aún así, recibimos un mensaje útil, también gratis.

### 3 Introducción a los argumentos posicionales

Un ejemplo:

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

Y ejecutando el código:

```
$ 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
```

Aquí está lo que está sucediendo:

- Hemos agregado el método `add_argument()`, el cual es el que usamos para especificar cuales opciones de la línea de comandos el programa está dispuesto a aceptar. En este caso, lo he llamado `echo` para que concuerde con su función.
- Llamar nuestro programa ahora requiere que especifiquemos una opción.
- El método `parse_args()` retorna de hecho algunos datos de las opciones especificadas, en este caso, `echo`.
- La variable es una forma de “magia” que `argparse` se realiza de forma gratuita (es decir, no es necesario especificar en qué variable se almacena ese valor). También notará que su nombre coincide con el argumento de cadena dado al método, `echo`.

Sin embargo, tenga en cuenta que, aunque la pantalla de ayuda luce bien y todo, en realidad no es tan útil como podría ser. Por ejemplo, vemos que tenemos `echo` como un argumento posicional, pero no sabemos lo que hace, de otra manera que no sea adivinar o leer el código fuente. Entonces, vamos a hacerlo un poco más útil:

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

Y la salida:

```
$ 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
```

Ahora, que tal si hacemos algo más útil:

```
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)
```

Lo siguiente es el resultado de ejecutar el código:

```
$ 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'
```

Eso no fue tan bien. Esto es porque `argparse` trata las opciones que le damos como cadenas, a menos que le digamos otra cosa. Entonces, vamos a llamar a `argparse` para tratar esa entrada como un entero:

```
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)
```

Lo siguiente es el resultado de ejecutar el código:

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

Eso fue bien. El programa ahora aún se cierra útilmente en caso de una entrada ilegal incorrecta antes de proceder.

## 4 Introducción a los argumentos opcionales

Hasta ahora hemos estado jugando con argumentos posicionales. Vamos a darle una mirada a como agregar los opcionales:

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

Y la salida:

```
$ 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
```

Esto es lo que está pasando:

- El programa está escrito para mostrar algo cuando `--verbosity` sea especificado y no mostrar nada cuando no.
- Para mostrar que la opción es de hecho opcional, no hay ningún error al ejecutar el programa sin ella. Tenga en cuenta que por defecto, si un argumento opcional no es usado, la variable relevante, en este caso `args.verbosity`, se le da `None` como valor, razón por la cual falla la prueba de verdad del `if`.
- El mensaje de ayuda es un poco diferente.
- Cuando usamos la opción `--verbosity`, también se debe especificar un valor, cualquier valor.

El ejemplo anterior acepta arbitrariamente valores enteros para `--verbosity`, pero para nuestro simple programa, solo dos valores son realmente útiles, `True` o `False`. Modifiquemos el código de acuerdo a esto:

```
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")
```

Y la salida:

```
$ 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
```

Esto es lo que está pasando:

- La opción ahora es más una bandera que algo que requiere un valor. Incluso cambiamos el nombre de la opción para que coincida con esa idea. Tenga en cuenta que ahora especificamos una nueva palabra clave, `action`, y le dimos el valor `"store_true"`. Esto significa que, si la opción es especificada, se asigna el valor `True` a `args.verbose`. No especificarlo implica `False`.
- Se queja cuando se especifica un valor, en verdadero espíritu de lo que realmente son los flags.
- Observe los diferentes textos de ayuda.

## 4.1 Opciones cortas

Si estas familiarizado con el uso de la línea de comandos, podrás observar que aún no he tocado el tema de las versiones cortas de las opciones. Es bastante simple:

```
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")
```

Y aquí va:

```
$ 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
```

Tenga en cuenta que la nueva habilidad es también reflejada en el texto de ayuda.

## 5 Combinar argumentos opcionales y posicionales

Nuestro programa sigue creciendo en complejidad:

```
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)
```

Y ahora la salida:

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

(continúe en la próxima página)

```
$ 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
```

- Hemos traído de vuelta un argumento posicional, de ahí la queja.
- Tenga en cuenta que el orden no importa.

Que tal si le retornamos a nuestro programa la capacidad de tener múltiples valores de verbosidad, y realmente usarlos:

```
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)
```

Y la salida:

```
$ 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
```

Todos estos se ven bien, excepto el último, que expone un error en nuestro programa. Corrijamos esto restringiendo los valores que la opción `--verbosity` puede aceptar:

```
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}")
```

(continúe en la próxima página)

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

Y la salida:

```
$ 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
```

Tenga en cuenta que el cambio se refleja tanto en el mensaje de error como en la cadena de ayuda.

Ahora, usemos un enfoque diferente para jugar con la verbosidad, lo cual es bastante común. También coincide con la forma en que el ejecutable de CPython maneja su propio argumento de verbosidad (verifique el resultado de `python --help`):

```
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)
```

Hemos introducido otra acción, «count», para contar el número de apariciones de opciones específicas.

```
$ 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:
```

(continúe en la próxima página)



```

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

```

- Si, ahora es mas una bandera (similar a `action="store_true"`) en la versión anterior de nuestro script. Esto debería explicar la queja.
- También se comporta de manera similar a la acción `"store_true"`.
- Ahora aquí una demostración de lo que la acción `"count"` da. Probablemente haya visto esta clase de uso antes.
- Y si no especificas la bandera `-v`, se considera que esa bandera tiene el valor `None`.
- Como debería esperarse, especificando la forma larga de la bandera, obtendríamos el mismo resultado.
- Lamentablemente, nuestra salida de ayuda no es muy informativa sobre la nueva capacidad que ha adquirido nuestro script, pero eso siempre se puede solucionar mejorando la documentación de nuestro script (por ejemplo, a través del argumento de la palabra clave `help`).
- La última salida expone un error en nuestro programa.

Vamos a arreglarlo:

```

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)

```

Y esto es lo que da:

```

$ 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'

```

- La primer salida fue correcta, y corrigió el error que teníamos antes. Es decir, queremos que cualquier valor `>= 2` sea lo más detallado posible.
- Tercer salida no tan buena.

Vamos a arreglar ese error:

```
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)
```

Acabamos de introducir otra palabra clave, `default`. Lo hemos configurado en 0 para que sea comparable con otros valores `int`. Recuerde que por defecto, si un argumento opcional no es especificado, obtiene el valor `None`, y eso no puede ser comparado con un valor `int` (de ahí la excepción `TypeError`).

Y:

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

Tu puedes llegar bastante lejos con lo que hemos aprendido hasta ahora, y solo arañado la superficie. El módulo `argparse` es muy poderoso, y exploraremos un poco mas antes de finalizar este tutorial.

## 6 Un poco mas avanzado

Qué pasaría si quisiéramos expandir nuestro pequeño programa para que tenga otros poderes, no solo cuadrados:

```
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)
```

Salida:

```
$ 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
```

(continúe en la próxima página)

```
options:
  -h, --help            show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

Tenga en cuenta que hasta ahora hemos estado usando el nivel de verbosidad para *cambiar* el texto que se muestra. El siguiente ejemplo en lugar de usar nivel de verbosidad para mostrar *mas* texto en su lugar:

```
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)
```

Salida:

```
$ 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 Especificando argumentos ambiguos

Cuando hay ambigüedad al decidir si un argumento es posicional o para un argumento, -- se puede usar para indicar `parse_args()` que todo lo que sigue a eso es un argumento posicional:

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

>>> # ambiguo, parse_args asume que es una opción
>>> 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)

>>> # ambiguo, la opción -n option acepta argumentos ávidamente
>>> 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 Opciones conflictivas

Hasta ahora, hemos estado trabajando con dos métodos de una instancia de `argparse.ArgumentParser`. Vamos a introducir un tercer método, `add_mutually_exclusive_group()`. Nos permite especificar opciones que entran en conflicto entre sí. Cambiemos también el resto del programa para que la nueva funcionalidad tenga mas sentido: presentamos la opción `--quiet`, la cual es lo opuesto a la opción `--verbose`:

```
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}")
```

Nuestro programa ahora es mas simple, y perdimos algunas funcionalidades en aras de la demostración. De todos modos, aquí esta el resultado:

```
$ 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
```

Esto debería ser sencillo de seguir. He agregado esa última salida para que se pueda ver el tipo de flexibilidad que obtiene, es decir, mezclar opciones de forma larga con opciones de forma corta.

Antes de concluir, probablemente quiera contarle a sus usuarios el propósito principal de su programa, solo en caso de que no lo supieran:

```
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:
```

(continúe en la próxima página)

(proviene de la página anterior)

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

Tenga en cuenta la ligera diferencia en el uso del texto. Tenga en cuenta `[-v | -q]`, lo cual nos indica que podemos usar `-v` o `-q`, pero no ambos al mismo tiempo:

```
$ 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 Cómo traducir la salida de argparse

La salida del módulo `argparse`, como su texto de ayuda y mensajes de error, se pueden traducir usando el módulo `gettext`. Esto permite que las aplicaciones localicen fácilmente los mensajes producidos por `argparse`. Consulte [i18n-howto](#) para más información.

Por ejemplo, en esta salida `argparse`:

```
$ 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
```

Las cadenas de caracteres `usage:`, `positional arguments:`, `options:` y `show this help message and exit`` son todas traducibles.

Para traducir estas cadenas de caracteres, primero se deben extraer en un archivo `.po`. Por ejemplo, usando [Babel](#), ejecute este comando:

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

Este comando extraerá todas las cadenas de caracteres traducibles del módulo `argparse` y las generará en un archivo llamado `messages.po`. Este comando asume que su instalación de Python está en `/usr/lib`.

Puede encontrar la ubicación del módulo `argparse` en su sistema usando este script:

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

Una vez que los mensajes en el archivo `.po` estén traducidos y las traducciones estén instaladas usando `gettext`, `argparse` podrá mostrar los mensajes traducidos.

Para traducir sus propias cadenas de caracteres en la salida `argparse`, 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)
```

Salida:

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

In this example, we:

- 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 Conclusión

El módulo `argparse` ofrece mucho más que solo lo mostrado aquí. Su documentación es bastante detallada y completa, y está llena de ejemplos. Habiendo seguido este tutorial, debe digerirlos fácilmente sin sentirse abrumado.