

---

# Tutoriel argparse

Version 3.13.2

Guido van Rossum and the Python development team

février 25, 2025

Python Software Foundation  
Email : docs@python.org

## Table des matières

<b>1</b>	<b>Concepts</b>	<b>2</b>
<b>2</b>	<b>Les bases</b>	<b>2</b>
<b>3</b>	<b>Introduction aux arguments positionnels</b>	<b>3</b>
<b>4</b>	<b>Introduction aux arguments optionnels</b>	<b>4</b>
4.1	Les paramètres raccourcis . . . . .	6
<b>5</b>	<b>Combinaison d'arguments positionnels et optionnels</b>	<b>6</b>
<b>6</b>	<b>Aller un peu plus loin</b>	<b>10</b>
6.1	Specifying ambiguous arguments . . . . .	11
6.2	Paramètres en conflit . . . . .	11
<b>7</b>	<b>How to translate the argparse output</b>	<b>13</b>
<b>8</b>	<b>Custom type converters</b>	<b>14</b>
<b>9</b>	<b>Conclusion</b>	<b>14</b>

---

### auteur

Tshepang Mbambo

Ce tutoriel est destiné à être une introduction en douceur à `argparse`, le module d'analyse de ligne de commande recommandé dans la bibliothèque standard de Python.

### Note

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 Concepts

Commençons par l'utilisation de la commande `ls` pour voir le type de fonctionnalité que nous allons étudier dans ce tutoriel d'introduction :

```
$ 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.
...
```

Quelques concepts que l'on peut apprendre avec les quatre commandes :

- La commande `ls` est utile quand elle est exécutée sans aucun paramètre. Elle affiche par défaut le contenu du dossier courant.
- Si l'on veut plus que ce qui est proposé par défaut, il faut l'indiquer. Dans le cas présent, on veut afficher un dossier différent : `pypy`. Ce que l'on a fait c'est spécifier un argument positionnel. C'est appelé ainsi car cela permet au programme de savoir quoi faire avec la valeur en se basant seulement sur sa position dans la ligne de commande. Ce concept est plus pertinent pour une commande comme `cp` dont l'usage de base est `cp SRC DEST`. Le premier argument est *ce que vous voulez copier* et le second est *où vous voulez le copier*.
- Maintenant, supposons que l'on veuille changer la façon dont le programme agit. Dans notre exemple, on affiche plus d'information pour chaque fichier que simplement leur nom. Dans ce cas, `-l` est un argument facultatif.
- C'est un fragment du texte d'aide. Cela peut être très utile quand on tombe sur un programme que l'on à jamais utilisé auparavant car on peut comprendre son fonctionnement simplement en lisant l'aide associée.

## 2 Les bases

Commençons par un exemple très simple qui ne fait (quasiment) rien :

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

Ce qui suit est le résultat de l'exécution du code :

```
$ 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
```

(suite sur la page suivante)

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

Voilà ce qu'il se passe :

- Exécuter le script sans aucun paramètre a pour effet de ne rien afficher sur la sortie d'erreur. Ce n'est pas très utile.
- Le deuxième commence à montrer l'intérêt du module `argparse`. On n'a quasiment rien fait mais on a déjà un beau message d'aide.
- L'option `--help`, que l'on peut aussi raccourcir en `-h`, est la seule option que l'on a gratuitement (pas besoin de la préciser). Préciser quoi que ce soit d'autre entraîne une erreur. Mais même dans ce cas, on reçoit aussi un message utile, toujours gratuitement.

### 3 Introduction aux arguments positionnels

Un exemple :

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

On exécute le code :

```
$ 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
```

Voilà ce qu'il se passe :

- We've added the `add_argument()` method, which is what we use to specify which command-line options the program is willing to accept. In this case, I've named it `echo` so that it's in line with its function.
- Utiliser le programme nécessite maintenant que l'on précise un paramètre.
- The `parse_args()` method actually returns some data from the options specified, in this case, `echo`.
- La variable est comme une forme de 'magie' que `argparse` effectue gratuitement (c.-à-d. pas besoin de préciser dans quelle variable la valeur est stockée). Vous aurez aussi remarqué que le nom est le même que l'argument en chaîne de caractères donné à la méthode : `echo`.

Notez cependant que, même si l'affichage d'aide paraît bien, il n'est pas aussi utile qu'il pourrait l'être. Par exemple, on peut lire que `echo` est un argument positionnel mais on ne peut pas savoir ce que cela fait autrement qu'en le devinant ou en lisant le code source. Donc, rendons-le un peu plus utile :

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

Et on obtient :

```
$ 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
```

À présent, que diriez-vous de faire quelque chose d'encore plus utile :

```
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)
```

Ce qui suit est le résultat de l'exécution du code :

```
$ 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'
```

Cela n'a pas très bien fonctionné. C'est parce que `argparse` traite les paramètres que l'on donne comme des chaînes de caractères à moins qu'on ne lui indique de faire autrement. Donc, disons à `argparse` de traiter cette entrée comme un entier :

```
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)
```

Ce qui suit est le résultat de l'exécution du code :

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

Cela a bien fonctionné. Maintenant le programme va même s'arrêter si l'entrée n'est pas légale avant de procéder à l'exécution.

## 4 Introduction aux arguments optionnels

Jusqu'à maintenant, on a joué avec les arguments positionnels. Regardons comment ajouter des paramètres optionnels :

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

Et le résultat :

```
$ 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
```

Voilà ce qu'il se passe :

- Le programme est écrit de sorte qu'il n'affiche rien sauf si l'option `--verbosity` est précisée.
- To show that the option is actually optional, there is no error when running the program without it. Note that by default, if an optional argument isn't used, the relevant variable, in this case `args.verbosity`, is given `None` as a value, which is the reason it fails the truth test of the `if` statement.
- Le message d'aide est quelque peu différent.
- Quand on utilise l'option `--verbosity` on doit aussi préciser une valeur, n'importe laquelle.

L'exemple ci-dessus accepte des valeurs entières arbitraires pour `--verbosity` mais pour notre programme simple seule deux valeurs sont réellement utiles : `True` et `False`. Modifions le code en accord avec cela :

```
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")
```

Et le résultat :

```
$ 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
```

Voilà ce qu'il se passe :

- The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, `action`, and give it the value `"store_true"`. This means that, if the option is specified, assign the value `True` to `args.verbose`. Not specifying it implies `False`.
- Dans l'esprit de ce que sont vraiment les options, pas des paramètres, il se plaint quand vous tentez de préciser une valeur.
- Notez que l'aide est différente.

## 4.1 Les paramètres raccourcis

Si vous êtes familier avec l'utilisation de la ligne de commande, vous avez dû remarquer que je n'ai pour l'instant rien dit au sujet des versions raccourcies des paramètres. C'est très 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")
```

Et voilà :

```
$ 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
```

Notez que la nouvelle option est aussi indiquée dans l'aide.

## 5 Combinaison d'arguments positionnels et optionnels

Notre programme continue de croître en complexité :

```
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)
```

Et voilà le résultat :

```
$ 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
```

- Nous avons ajouté un argument nommé, d'où le message d'erreur.
- Notez que l'ordre importe peu.

Qu'en est-il si nous donnons à ce programme la possibilité d'avoir plusieurs niveaux de verbosité, et que celui-ci les prend en compte :

```
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)
```

Et le résultat :

```
$ 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
```

Tout semble bon sauf le dernier, qui montre que notre programme contient un bogue. Corrigons cela en restreignant les valeurs que `--verbosity` accepte :

```
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)
```

Et le résultat :

```
$ 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
```

(suite sur la page suivante)

```
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
```

Notez que ce changement est pris en compte à la fois dans le message d'erreur et dans le texte d'aide.

Essayons maintenant une approche différente pour jouer sur la verbosité, ce qui arrive fréquemment. Cela correspond également à comment le programme CPython gère ses propres paramètres de verbosité (jetez un œil sur la sortie de la commande `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)
```

Nous avons introduit une autre action, "count", pour compter le nombre d'occurrences d'une option en particulier :

```
$ 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
```

- Oui, c'est maintenant d'avantage une option (similaire à `action="store_true"`) de la version précédente de notre script. Cela devrait expliquer le message d'erreur.
- Cela se comporte de la même manière que l'action "store\_true".



- Maintenant voici une démonstration de ce que l'action "count" fait. Vous avez sûrement vu ce genre d'utilisation auparavant.
- Et si vous ne spécifiez pas l'option -v, cette option prendra la valeur None.
- Comme on s'y attend, en spécifiant l'option dans sa forme longue, on devrait obtenir la même sortie.
- Malheureusement, notre sortie d'aide n'est pas très informative à propos des nouvelles possibilités de notre programme, mais cela peut toujours être corrigé en améliorant sa documentation (en utilisant l'argument help).
- La dernière sortie du programme montre que celui-ci contient un bogue.

Corrigeons :

```
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)
```

Et c'est ce que ça donne :

```
$ 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'
```

- Les premières exécutions du programme sont correctes, et le bogue que nous avons eu est corrigé. Cela dit, nous voulons que n'importe quelle valeur `>= 2` rende le programme aussi verbeux que possible.
- La troisième sortie de programme n'est pas si bien que ça.

Corrigeons ce bogue :

```
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:
```

(suite sur la page suivante)

```
print(answer)
```

Nous venons juste d'introduire un nouveau mot clef, `default`. Nous l'avons défini à 0 pour le rendre comparable aux autres valeurs. Rappelez-vous que par défaut, si un argument optionnel n'est pas spécifié, il sera défini à `None`, et ne pourra pas être comparé à une valeur de type entier (une erreur `TypeError` serait alors levée).

Et :

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

Vous pouvez aller assez loin seulement avec ce que nous avons appris jusqu'à maintenant, et nous n'avons qu'aperçu la surface. Le module `argparse` est très puissant, et nous allons l'explorer un peu plus avant la fin de ce tutoriel.

## 6 Aller un peu plus loin

Qu'en est-il si nous souhaitons étendre notre mini programme pour le rendre capable de calculer d'autres puissances, et pas seulement des carrés :

```
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)
```

Sortie :

```
$ 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 this help message and exit
$ python prog.py 4 2 -v
4^2 == 16
```

Il est à noter que jusqu'à présent nous avons utilisé le niveau de verbosité pour *changer* le texte qui est affiché. L'exemple suivant au contraire utilise le niveau de verbosité pour afficher *plus* de texte à la place :

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
```

(suite sur la page suivante)

```

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)

```

Sortie :

```

$ 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 Paramètres en conflit

So far, we have been working with two methods of an `argparse.ArgumentParser` instance. Let's introduce a third one, `add_mutually_exclusive_group()`. It allows for us to specify options that conflict with each other. Let's also change the rest of the program so that the new functionality makes more sense : we'll introduce the `--quiet` option, which will be the opposite of the `--verbose` one :

```

import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")

```

(suite sur la page suivante)

```

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}")

```

Notre programme est maintenant plus simple, et nous avons perdu des fonctionnalités pour faire cette démonstration. Peu importe, voici la sortie du programme :

```

$ 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

```

Cela devrait être facile à suivre. J'ai ajouté cette dernière sortie pour que vous puissiez voir le genre de flexibilité que vous pouvez avoir, par exemple pour faire un mélange entre des paramètres courts et longs.

Avant d'en finir, vous voudrez certainement dire à vos utilisateurs quel est le but principal de votre programme, juste dans le cas où ils ne le sauraient pas :

```

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}")

```

Notez cette nuance dans le texte d'utilisation. Les options `[-v | -q]` nous disent que nous pouvons utiliser au choix `-v` ou `-q`, mais pas les deux ensemble :

```
$ 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 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)
```

Sortie :

```
$ 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 Conclusion

Le module `argparse` offre bien plus que ce qui est montré ici. Sa documentation est assez détaillée, complète et pleine d'exemples. En ayant accompli ce tutoriel, vous pourrez facilement comprendre cette documentation sans vous sentir dépassé.