Tutoriel argparse

Version 3.8.18

Guido van Rossum
and the Python development team

février 08, 2024

Python Software Foundation
Email : docs@python.org

Table des matières

auteur Tshepang Lekhonkhobe

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 : Il y a deux autres modules qui remplissent le même rôle : `getopt` (un équivalent de `getopt()` du langage C) et `optparse` qui est obsolète. Il faut noter que `argparse` est basé sur `optparse` et donc s'utilise de manière très similaire.

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 ficher 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 :

```
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h]

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python3 prog.py foo
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 :

```
$ python3 prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python3 prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py foo
foo
```

Voilà ce qu'il se passe :

- On a ajouté la méthode `add_argument()` que l'on utilise pour préciser quels paramètre de lignes de commandes le programme peut accepter. Dans le cas présent, je l'ai appelé `echo` pour que cela corresponde à sa fonction.
- Utiliser le programme nécessite maintenant que l'on précise un paramètre.
- La méthode `parse_args()` renvoie en réalité certaines données des paramètres précisés, dans le cas présent : `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 :

```
$ python3 prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

optional arguments:
  -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 :

```
$ python3 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 :

```
$ python3 prog.py 4
16
$ python3 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 :

```
$ python3 prog.py --verbosity 1
verbosity turned on
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

optional arguments:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python3 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.
— Pour montrer que l'option est bien optionnelle il n'y aura pas d'erreur s'il on exécute le programme sans celle-ci. Notez que par défaut, si une option n'est pas utilisée, la variable associée, dans le cas présent : `args.verbosity`, prend la valeur `None` c'est pourquoi elle échoue le test de vérité de l'assertion `if`.
— 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 :

```
$ python3 prog.py --verbose
verbosity turned on
$ python3 prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python3 prog.py --help
usage: prog.py [-h] [--verbose]

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

Voilà ce qu'il se passe :

— Maintenant le paramètre est plus une option que quelque chose qui nécessite une valeur. On a même changé le nom du paramètre pour qu'il corresponde à cette idée. Notez que maintenant on précise une nouvelle `action` clavier et qu'on lui donne la valeur `"store_true"`. Cela signifie que si l'option est précisée la valeur `True` est assignée à `args.verbose`. Ne rien préciser implique la valeur `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 :

```
$ python3 prog.py -v
verbosity turned on
$ python3 prog.py --help
usage: prog.py [-h] [-v]

optional arguments:
  -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("the square of {} equals {}".format(args.square, answer))
else:
    print(answer)
```

Et voilà le résultat :

```
$ python3 prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python3 prog.py 4
16
$ python3 prog.py 4 --verbose
the square of 4 equals 16
$ python3 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("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

Et le résultat :

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python3 prog.py 4 -v 1
4^2 == 16
$ python3 prog.py 4 -v 2
the square of 4 equals 16
$ python3 prog.py 4 -v 3
16
```

Tout semble bon sauf le dernier, qui montre que notre programme contient un bogue. Corrigeons 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("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

Et le résultat :

```
$ python3 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)
$ python3 prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

optional arguments:
  -h, --help            show this help message and exit
  -v {0,1,2}, --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("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

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

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
4^2 == 16
$ python3 prog.py 4 -vv
the square of 4 equals 16
$ python3 prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python3 prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python3 prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python3 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("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

Et c'est ce que ça donne :

```
$ python3 prog.py 4 -vvv
the square of 4 equals 16
$ python3 prog.py 4 -vvvv
the square of 4 equals 16
$ python3 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("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

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

Et :

```
$ python3 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("{} to the power {} equals {}".format(args.x, args.y, answer))
elif args.verbosity >= 1:
    print("{}^{} == {}".format(args.x, args.y, answer))
else:
    print(answer)
```

Sortie :

```
$ python3 prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python3 prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python3 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")
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("Running '{}'".format(__file__))
if args.verbosity >= 1:
    print("{}^{} == ".format(args.x, args.y), end="")
print(answer)
```

Sortie :

```
$ python3 prog.py 4 2
16
$ python3 prog.py 4 2 -v
4^2 == 16
$ python3 prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

6.1 Paramètres en conflit

Jusque là, nous avons travaillé avec deux méthodes d'une instance de `argparse.ArgumentParser`. En voici une troisième, `add_mutually_exclusive_group()`. Elle nous permet de spécifier des paramètres qui sont en conflit entre eux. Changeons aussi le reste du programme de telle sorte que la nouvelle fonctionnalité fasse sens : nous allons introduire l'option `--quiet`, qui va avoir l'effet opposé de l'option `--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("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(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 :

```
$ python3 prog.py 4 2
4^2 == 16
$ python3 prog.py 4 2 -q
16
$ python3 prog.py 4 2 -v
4 to the power 2 equals 16
$ python3 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
$ python3 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 ou 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("{} to the power {} equals {}".format(args.x, args.y, answer))
```

(suite sur la page suivante)

(suite de la page précédente)

```
else:
    print("{}^{} == {}".format(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 :

```
$ python3 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

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

7 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.