
Tutorial de Argparse

Release 2.7.18

**Guido van Rossum
and the Python development team**

maio 07, 2020

**Python Software Foundation
Email: docs@python.org**

Sumário

1	Conceitos	2
2	O Básico	2
3	Introdução ao uso dos Argumentos Posicionais	3
4	Introducing Optional arguments	5
4.1	Opções Curtas	6
5	Combining Positional and Optional arguments	6
6	Getting a little more advanced	10
6.1	Opções Conflitantes	12
7	Conclusão	13

autor Tshepang Lekhonkhobe

This tutorial is intended to be a gentle introduction to `argparse`, the recommended command-line parsing module in the Python standard library. This was written for `argparse` in Python 3. A few details are different in 2.x, especially some exception messages, which were improved in 3.x.

Nota: Existem outros dois módulos que cumprem esta mesma tarefa, chamados `getopt` (equivalente ao `getopt()` da linguagem C) e outro que hoje está descontinuado `optparse`. Note também que o `argparse` é baseado no módulo `optparse`, e, portanto, possui bastante similaridade em termos de uso.

1 Conceitos

Demonstraremos o tipo de funcionalidade que vamos explorar neste tutorial introdutório fazendo uso do 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.
...
```

Alguns conceitos que podemos aprender a partir destes quatro comandos:

- O comando **ls** é útil quando usado sem nenhuma opção. Por padrão, ele mostra o conteúdo do diretório atual.
- If we want beyond what it provides by default, we tell it a bit more. In this case, we want it to display a different directory, `pypy`. What we did is specify what is known as a positional argument. It's named so because the program should know what to do with the value, solely based on where it appears on the command line. This concept is more relevant to a command like **cp**, whose most basic usage is `cp SRC DEST`. The first position is *what you want copied*, and the second position is *where you want it copied to*.
- Now, say we want to change behaviour of the program. In our example, we display more info for each file instead of just showing the file names. The `-l` in that case is known as an optional argument.
- Esse é um trecho do texto de ajuda. É muito útil que possas encontrar um programa que nunca usastes antes e poder descobrir como o mesmo funciona simplesmente lendo o seu texto de ajuda.

2 O Básico

Começemos com um exemplo muito simples que irá fazer (quase) nada:

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

Seguir temos o resultado da execução do código:

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

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

(continua na próxima página)

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

Eis aqui o que está acontecendo:

- Executar o script sem qualquer opção resultará que nada será exibido em stdout. Isso não é útil.
- O segundo começa a exibir as utilidades do módulo `argparse`. Não fizemos quase nada, mas já recebemos uma boa mensagem de ajuda.
- A opção `--help`, que também pode ser encurtada para `-h`, é a única opção que obtemos livremente (ou seja, não é necessário determiná-la). Determinar qualquer outra coisa resulta num erro. Mas mesmo assim, recebemos uma mensagem de utilização bastante útil, também de graça.

3 Introdução ao uso dos Argumentos Posicionais

Um exemplo:

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

E executando o 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

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

Aqui está o que acontecerá:

- Nós adicionamos o método `add_argument()`, cujo o mesmo usamos para especificar quais opções de linha de comando o programa está disposto a aceitar. Neste caso, eu o nomeei `echo` para que ele esteja de acordo com sua função.
- Chamar o nosso programa neste momento, requer a especificação de uma opção.
- O método `parse_args()` realmente retorna alguns dados das opções especificadas, neste caso, `echo`.
- The variable is some form of ‘magic’ that `argparse` performs for free (i.e. no need to specify which variable that value is stored in). You will also notice that its name matches the string argument given to the method, `echo`.

Note however that, although the help display looks nice and all, it currently is not as helpful as it can be. For example we see that we got `echo` as a positional argument, but we don’t know what it does, other than by guessing or by reading the source code. So, let’s make it a bit more useful:

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

E, iremos obter:

```
$ python 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
```

Now, how about doing something even more useful:

```
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
```

Seguir temos o resultado da execução do 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 **: 'str' and 'int'
```

That didn't go so well. That's because argparse treats the options we give it as strings, unless we tell it otherwise. So, let's tell argparse to treat that input as an integer:

```
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
```

Seguir temos o resultado da execução do código:

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

That went well. The program now even helpfully quits on bad illegal input before proceeding.

4 Introducing Optional arguments

So far we have been playing with positional arguments. Let us have a look on how to add optional ones:

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

E a saída:

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

optional arguments:
  -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
```

Eis aqui o que está acontecendo:

- The program is written so as to display something when `--verbosity` is specified and display nothing when not.
- 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.
- The help message is a bit different.
- When using the `--verbosity` option, one must also specify some value, any value.

The above example accepts arbitrary integer values for `--verbosity`, but for our simple program, only two values are actually useful, `True` or `False`. Let's modify the code accordingly:

```
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"
```

E a saída:

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

(continua na próxima página)

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

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

Eis aqui o que está acontecendo:

- 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`.
- It complains when you specify a value, in true spirit of what flags actually are.
- Notice the different help text.

4.1 Opções Curtas

If you are familiar with command line usage, you will notice that I haven't yet touched on the topic of short versions of the options. It's quite 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"
```

And here goes:

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

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

Note that the new ability is also reflected in the help text.

5 Combining Positional and Optional arguments

Our program keeps growing in complexity:

```
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",
```

(continua na próxima página)

```

        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

```

E agora o que foi enviado para a saída padrão.

```

$ 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

```

- We've brought back a positional argument, hence the complaint.
- Note that the order does not matter.

How about we give this program of ours back the ability to have multiple verbosity values, and actually get to use them:

```

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

```

E a saída:

```

$ 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

```

These all look good except the last one, which exposes a bug in our program. Let's fix it by restricting the values the `--verbosity` option can accept:

```

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

```

E a saída:

```

$ 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

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

```

Note that the change also reflects both in the error message as well as the help string.

Now, let's use a different approach of playing with verbosity, which is pretty common. It also matches the way the CPython executable handles its own verbosity argument (check the output of `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

```

We have introduced another action, "count", to count the number of occurrences of a specific optional arguments:

```

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

```

(continua na próxima página)


```

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

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

```

- Yes, it's now more of a flag (similar to `action="store_true"`) in the previous version of our script. That should explain the complaint.
- It also behaves similar to “store_true” action.
- Now here's a demonstration of what the “count” action gives. You've probably seen this sort of usage before.
- And, just like the “store_true” action, if you don't specify the `-v` flag, that flag is considered to have `None` value.
- As should be expected, specifying the long form of the flag, we should get the same output.
- Sadly, our help output isn't very informative on the new ability our script has acquired, but that can always be fixed by improving the documentation for our script (e.g. via the `help` keyword argument).
- That last output exposes a bug in our program.

Vamos corrigir:

```

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

```

E isso aqui é o mesmo retorna:

```

$ 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: unorderable types: NoneType() >= int()
```

- First output went well, and fixes the bug we had before. That is, we want any value ≥ 2 to be as verbose as possible.
- Third output not so good.

Vamos corrigir esse bug:

```
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
```

We've just introduced yet another keyword, `default`. We've set it to 0 in order to make it comparable to the other int values. Remember that by default, if an optional argument isn't specified, it gets the `None` value, and that cannot be compared to an int value (hence the `TypeError` exception).

E:

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

You can go quite far just with what we've learned so far, and we have only scratched the surface. The `argparse` module is very powerful, and we'll explore a bit more of it before we end this tutorial.

6 Getting a little more advanced

What if we wanted to expand our tiny program to perform other powers, not just squares:

```
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)
```

(continua na próxima página)

```
else:
    print answer
```

Saída:

```
$ 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

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

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

Notice that so far we've been using verbosity level to *change* the text that gets displayed. The following example instead uses verbosity level to display *more* text instead:

```
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),
print answer
```

Saída:

```
$ 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 Opções Conflitantes

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")
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)
```

Our program is now simpler, and we've lost some functionality for the sake of demonstration. Anyways, here's the output:

```
$ 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
```

That should be easy to follow. I've added that last output so you can see the sort of flexibility you get, i.e. mixing long form options with short form ones.

Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don't know:

```
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
```

(continua na próxima página)

```

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

```

Note that slight difference in the usage text. Note the `[-v | -q]`, which tells us that we can either use `-v` or `-q`, but not both at the same time:

```

$ 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

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

```

7 Conclusão

The `argparse` module offers a lot more than shown here. Its docs are quite detailed and thorough, and full of examples. Having gone through this tutorial, you should easily digest them without feeling overwhelmed.