

---

# Tutorial de Argparse

*Release 3.12.9*

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

abril 08, 2025

Python Software Foundation  
Email: docs@python.org

## Sumário

<b>1</b>	<b>Conceitos</b>	<b>2</b>
<b>2</b>	<b>O básico</b>	<b>2</b>
<b>3</b>	<b>Apresentando os argumentos posicionais</b>	<b>3</b>
<b>4</b>	<b>Apresentando os argumentos opcionais</b>	<b>4</b>
4.1	Opções curtas . . . . .	6
<b>5</b>	<b>Combinando argumentos posicionais e opcionais</b>	<b>6</b>
<b>6</b>	<b>Avançando um pouco mais</b>	<b>10</b>
6.1	Especificando argumentos ambíguos . . . . .	11
6.2	Opções conflitantes . . . . .	12
<b>7</b>	<b>Como traduzir a saída do argparse</b>	<b>13</b>
<b>8</b>	<b>Conversores de tipo personalizados</b>	<b>14</b>
<b>9</b>	<b>Conclusão</b>	<b>14</b>

---

### autor

Tshepang Mbambo

Este tutorial pretende ser uma introdução gentil ao `argparse` — o módulo recomendado na biblioteca padrão do Python para fazer a análise de linha de comando.

### 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
Uso: ls [OPÇÃO]... [ARQUIVO]...
Lista informações sobre os ARQUIVOS (no diretório atual por padrão).
Lista as entradas em ordem alfabética se não for usada nenhuma opção -cftuvSUX
nem --sort..
...
```

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.
- Se quisermos além do que ele fornece por padrão, contamos um pouco mais. Neste caso, queremos que ele exiba um diretório diferente, `pypy`. O que fizemos foi especificar o que é conhecido como argumento posicional. Ele é chamado assim porque o programa deve saber o que fazer com o valor, apenas com base em onde ele aparece na linha de comando. Este conceito é mais relevante para um comando como **cp**, cujo uso mais básico é `cp SRC DEST`. A primeira posição é *o que você deseja copiar* e a segunda posição é *para onde você deseja copiar*.
- Agora, digamos que queremos mudar o comportamento do programa. Em nosso exemplo, exibimos mais informações para cada arquivo em vez de apenas mostrar os nomes dos arquivos. O `-l` nesse caso é conhecido como um argumento opcional.
- 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()
```

A seguir, temos o resultado da execução do 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]
```

(continua na próxima página)

```
prog.py: error: unrecognized arguments: --verbose
$ 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 uso bastante útil, também de graça.

### 3 Apresentando os 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

options:
  -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`.
- A variável é uma forma de “mágica” que `argparse` executa de brinde (ou seja, não é necessário especificar em qual variável esse valor é armazenado). Você também notará que seu nome corresponde ao argumento string dado ao método, `echo`.

Observe, no entanto, que, embora a tela de ajuda pareça boa e tudo, atualmente não é tão útil quanto poderia ser. Por exemplo, vemos que temos `echo` como um argumento posicional, mas não sabemos o que ele faz, além de adivinhar ou ler o código-fonte. Então, vamos torná-lo um pouco mais útil:

```
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

options:
  -h, --help          show this help message and exit
```

Agora, que tal fazer algo ainda mais 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)
```

A 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 ** or pow(): 'str' and 'int'
```

Isso não correu tão bem. Isso porque `argparse` trata as opções que damos a ele como strings, a menos que digamos o contrário. Então, vamos dizer ao `argparse` para tratar essa entrada como um inteiro:

```
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)
```

A 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'
```

Correu bem. O programa agora até fecha com ajuda de entrada ilegal ruim antes de prosseguir.

## 4 Apresentando os argumentos opcionais

Até agora, jogamos com argumentos posicionais. Vamos dar uma olhada em como adicionar opcionais:

```
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]

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
```

Eis aqui o que está acontecendo:

- O programa é escrito de forma a exibir algo quando `--verbosity` é especificado e não exibir nada quando não for.
- Para mostrar que a opção é realmente opcional, não há erro ao executar o programa sem ela. Observe que, por padrão, se um argumento opcional não for usado, a variável relevante, neste caso `args.verbosity`, recebe `None` como valor, razão pela qual falha no teste de verdade da instrução `if`.
- A mensagem de ajuda é um pouco diferente.
- Ao usar a opção `--verbosity`, deve-se também especificar algum valor, qualquer valor.

O exemplo acima aceita valores inteiros arbitrários para `--verbosity`, mas para nosso programa simples, apenas dois valores são realmente úteis, `True` ou `False`. Vamos modificar o código de acordo:

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

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

Eis aqui o que está acontecendo:

- A opção agora é mais um sinalizador do que algo que requer um valor. Até mudamos o nome da opção para corresponder a essa ideia. Observe que agora especificamos uma nova palavra reservada, `action`, e damos a ela o valor `"store_true"`. Isso significa que, se a opção for especificada, atribui o valor `True` para `args.verbose`. Não especificá-la implica em `False`.
- Ele reclama quando você especifica um valor, no verdadeiro espírito do que os sinalizadores realmente são.
- Observe o texto de ajuda diferente.

## 4.1 Opções curtas

Se você estiver familiarizado com o uso da linha de comando, notará que ainda não toquei no tópico das versões curtas das opções. É bem simples:

```
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")
```

E aqui vai:

```
$ 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
```

Observe que a nova habilidade também é refletida no texto de ajuda.

## 5 Combinando argumentos posicionais e opcionais

Nosso programa continua crescendo em complexidade:

```
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)
```

E agora a saída:

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

(continua na 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
```

- Trouxemos de volta um argumento posicional, daí a reclamação.
- Observe que a ordem não importa.

Que tal devolvermos a este nosso programa a capacidade de ter vários valores de verbosidade e realmente usá-los:

```
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)
```

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
```

Todos eles parecem bons, exceto o último, que expõe um bug em nosso programa. Vamos corrigi-lo restringindo os valores que a opção `--verbosity` pode aceitar:

```
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}")
```

(continua na próxima página)

```
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

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

Observe que a alteração também reflete tanto na mensagem de erro quanto na string de ajuda.

Agora, vamos usar uma abordagem diferente de brincar com a verbosidade, que é bastante comum. Ele também corresponde à maneira como o executável do CPython trata seu próprio argumento de verbosidade (verifique a saída 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)
```

Introduzimos outra ação, “contar”, para contar o número de ocorrências de opções 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:
```

(continua na 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

```

- Sim, agora é mais um sinalizador (semelhante a `action="store_true"`) na versão anterior do nosso script. Isso deve explicar a reclamação.
- Ele também se comporta de maneira semelhante à ação “store\_true”.
- Agora aqui está uma demonstração do que a ação “contar” oferece. Você provavelmente já viu esse tipo de uso antes.
- E se você não especificar o sinalizador `-v`, esse sinalizador será considerado como tendo valor `None`.
- Como deve ser esperado, especificando a forma longa do sinalizador, devemos obter a mesma saída.
- Infelizmente, nossa saída de ajuda não é muito informativa sobre a nova habilidade que nosso script adquiriu, mas isso sempre pode ser corrigido melhorando a documentação de nosso script (por exemplo, através do argumento nomeado `help`).
- Essa última saída expõe um bug em nosso programa.

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(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {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: '>=' not supported between instances of 'NoneType' and 'int'

```

- A primeira saída correu bem e corrige o bug que tínhamos antes. Ou seja, queremos que qualquer valor `>= 2` seja o mais detalhado possível.
- A terceira saída não está tão boa.

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(f"the square of {args.square} equals {answer}")
elif args.verbosity >= 1:
    print(f"{args.square}^2 == {answer}")
else:
    print(answer)
```

Acabamos de introduzir outra palavra reservada, `default`. Nós o configuramos como 0 para torná-lo comparável aos outros valores `int`. Lembre-se que por padrão, se um argumento opcional não for especificado, ele obtém o valor `None`, e isso não pode ser comparado a um valor `int` (daí a exceção `TypeError`).

E:

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

Você pode ir muito longe apenas com o que aprendemos até agora, e nós apenas arranhamos a superfície. O módulo `argparse` é muito poderoso, e vamos explorar um pouco mais antes de terminar este tutorial.

## 6 Avançando um pouco mais

E se quiséssemos expandir nosso pequeno programa, ampliando seu potencial:

```
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)
```

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
```

(continua na próxima página)

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

Observe que até agora estamos usando o nível de verbosidade para *alterar* o texto que é exibido. O exemplo a seguir usa o nível de verbosidade para exibir *mais* texto:

```
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)
```

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 Especificando argumentos ambíguos

Quando há ambiguidade em decidir se um argumento é posicional ou para um argumento, -- pode ser usado para dizer `parse_args()` que tudo depois disso é um argumento posicional:

```
>>> 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 Opções conflitantes

Até agora, trabalhamos com dois métodos de uma instância `argparse.ArgumentParser`. Vamos apresentar um terceiro, `add_mutually_exclusive_group()`. Ele nos permite especificar opções que entram em conflito umas com as outras. Vamos também alterar o resto do programa para que a nova funcionalidade faça mais sentido: vamos introduzir a opção `--quiet`, que será o oposto da opção `--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}")
```

Nosso programa agora está mais simples e perdemos algumas funcionalidades para demonstração. De qualquer forma, aqui está a saída:

```
$ 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
```

Isso deve ser fácil de seguir. Eu adicionei essa última saída para que você possa ver o tipo de flexibilidade que obtém, ou seja, misturar opções de formato longo com formatos curtos.

Antes de concluirmos, você provavelmente quer dizer aos seus usuários o propósito principal do seu programa, caso eles não saibam:

```
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:
```

(continua na próxima página)

```

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

```

Observe essa pequena diferença no texto de uso. Observe o `[-v | -q]`, que nos diz que podemos usar `-v` ou `-q`, mas não ambos ao mesmo tempo:

```

$ 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 Como traduzir a saída do argparse

A saída do módulo `argparse`, como seu texto de ajuda e mensagens de erro, são todos traduzidos usando o módulo `gettext`. Isso permite que os aplicativos localizem facilmente as mensagens produzidas por `argparse`. Veja também [i18n-howto](#).

Por exemplo, nesta saída `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

```

As strings `usage:`, `positional arguments:`, `options:` e `show this help message and exit` são todos traduzíveis.

Para traduzir essas strings, elas devem primeiro ser extraídas em um arquivo `.po`. Por exemplo, usando [Babel](#), execute este comando:

```

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

```

Este comando extrairá todas as strings traduzíveis do módulo `argparse` e as enviará para um arquivo chamado `messages.po`. Este comando presume que sua instalação do Python está em `/usr/lib`.

Você pode descobrir a localização do módulo `argparse` em seu sistema usando este script:

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

Uma vez que as mensagens no arquivo `.po` sejam traduzidas e as traduções instaladas usando `gettext`, `argparse` será capaz de exibir as mensagens traduzidas.

Para traduzir suas próprias strings na saída `argparse`, use `gettext`.

## 8 Conversores de tipo personalizados

O módulo `argparse` permite que você especifique conversores de tipo personalizados para seus argumentos de linha de comando. Isso permite modificar a entrada de usuário antes de armazená-la na `argparse.Namespace`. Isso pode ser útil quando você precisa pré-processar a entrada antes dela ser usada em seu programa.

Ao usar um conversor de tipo personalizado, você pode usar qualquer chamável que recebe uma única string como argumento (o valor do argumento) e retorna o valor convertido. Porém, se você precisa tratar de cenários mais complexos, você pode usar uma classe de ação personalizada com o parâmetro **action**.

Por exemplo, digamos que você deseja tratar de argumentos com diferentes prefixos e processá-los adequadamente:

```
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)
```

Saída:

```
$ python prog.py -a valor1 +a valor2
Namespace(a=[('-', 'valor1'), ('+', 'valor2')])
```

Neste exemplo, nós:

- Criamos um analisador sintático com caracteres de prefixo personalizado usando parâmetro `prefix_chars`.
- Definimos dois argumentos, `-a` e `+a`, que usaram o parâmetro `type` para criar conversores de tipo personalizados para armazenar o valor em uma tupla com o prefixo.

Sem os conversores de tipo personalizados, os argumentos tratariam `-a` e `+a` como o mesmo argumento, o que não era o desejado. Ao usar conversores de tipo personalizados, conseguimos diferenciar entre dois argumentos.

## 9 Conclusão

O módulo `argparse` oferece muito mais do que o mostrado aqui. Sua documentação é bastante detalhada e completa, e cheia de exemplos. Tendo passado por este tutorial, você deve digeri-la facilmente sem se sentir sobrecarregado.