

---

# Argparse チュートリアル

リリース 3.13.5

Guido van Rossum and the Python development team

8 月 05, 2025

## 目次

1	コンセプト	2
2	基礎	3
3	位置引数の入門	3
4	Optional 引数の導入	6
4.1	短いオプション . . . . .	7
5	位置引数と Optional 引数の併用	8
6	もうちょっとだけ学ぶ	13
6.1	多義性のある引数の指定 . . . . .	14
6.2	競合するオプション . . . . .	15
7	argparse の出力を翻訳するには	17
8	Custom type converters	18
9	結び	19

---

author

Tshepang Mbambo

このチュートリアルでは、`argparse` を丁寧に説明します。`argparse` は、Python 標準ライブラリの一部であり、おすすめのコマンドライン引数の解析モジュールです。

 注釈

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 コンセプト

`ls` コマンドを使って、このチュートリアルで私たちが学ぶ機能をいくつか見てみましょう:

```
$ 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.
...
```

上の4つの実行結果から、以下のことが分かります:

- `ls` コマンドは、まったくオプションを指定せずに実行したとしても役に立ちます。デフォルトの動作は、カレントディレクトリの内容を表示することです。
- デフォルトの動作で提供される以上のことをしたい場合、すこしだけオプションを指定する必要があります。別のディレクトリ `pypy` を表示したい場合、私たちがしたことは、位置引数として知られる引数を指定することです。これは、引数がコマンドラインのどの位置に現れたかということだけを基に、プログラムがその値について何をやるのか分かるべきなので、このように名づけられています。このコンセプトは `cp` のようなコマンドで重要な意味があります。`cp` コマンドのもっとも基本的な使い方は、`cp SRC DEST` です。最初の引数は **何をコピーしたいか** であり、二つ目の引数は **どこにコピーしたいか** を意味します。
- プログラムの振る舞いを変えましょう。例では、単にファイル名を表示する代わりにそれぞれのファイルに関する多くの情報を表示します。このケースでは、`-l` は optional 引数として知られます。

- ヘルプテキストの抜粋です。この実行の仕方は、まだ使用したことがないプログラムにたいして行うと有用で、ヘルプテキストを読むことで、プログラムがどのように動作するかわかります。

## 2 基礎

(ほとんど) 何もしない、とても簡単な例から始めましょう:

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

下記がこのコードを実行した結果です:

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

こんなことが起こりました:

- オプションなしでスクリプトを実行した結果、なにも標準出力に表示されませんでした。それほど便利ではありませんね。
- 二つ目の実行結果から `argparse` モジュールの有用性がわかります。ほとんど何もしていないのに、すてきなヘルプメッセージが手に入りました。
- `--help` (これは `-h` と短縮できます) だけが無料のオプションです (つまりプログラムで指示する必要はありません)。他のオプションを指定するとエラーになります。エラー時の有用な用法メッセージも、プログラムで指示することなく出力できます。

## 3 位置引数の入門

以下に例を示します:

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

(次のページに続く)

```
args = parser.parse_args()
print(args.echo)
```

このコードを実行してみましょう:

```
$ 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
```

こんなことが起こりました:

- プログラムが受け付けるコマンドラインオプションを指定するメソッドである `add_argument()` を追加しました。ここでは、その機能にあわせて引数名を `echo` としました。
- プログラムを実行すると、オプションを指定するように要求されます。
- `parse_args()` メソッドは指定されたオプションを、この場合は `echo` として、返します。
- この変数は `argparse` が自動的に行うある種の魔法です（つまり、値を格納する変数を指定する必要がありません）。変数の名前がメソッドに与えた文字列引数 `echo` と同じことに気付いたでしょう。

ヘルプメッセージは十分のように見えますが、それほど分かりやすくありません。たとえば、`echo` は位置引数であることがわかりますが、それが何であるかを知るためには推測するかソースコードを見なければなりません。もうすこしヘルプメッセージ分かりやすくしてみましょう:

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

修正した結果は以下のようになります:

```
$ 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
```

次は、もっと有益なことをしてみませんか？:

```
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)
```

下記がこのコードを実行した結果です:

```
$ 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'
```

上手くいきませんでした。何も伝えなければ、`argparse` は与えられたオプションを文字列として扱います。`argparse` にオプションの値を整数として扱うように伝えましょう:

```
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)
```

下記がこのコードを実行した結果です:

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

今度は上手くいきました。このプログラムは不正な入力を与えられるとそれを処理せずに、より親切なメッセージを表示して実行を終了します。

## 4 Optional 引数の導入

ここまで位置引数を扱ってきました。optional 引数を追加する方法についても見ていきましょう:

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

実行してみましょう:

```
$ 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
```

こんなことが起こりました:

- プログラムは、`--verbosity` が指定された場合はなにかしらを表示し、指定されなければ何も表示をしないように書かれています。
- オプションの指定が実際に任意であることを示すために、プログラムをオプション指定なしで実行してもエラーにはなっていません。オプション引数が指定されなかった場合、関連する変数、この例では `args.verbosity`、の値にはデフォルトで `None` がセットされます。これが `if` 文による真偽テストに失敗した理由です。
- ヘルプメッセージが少し変わりました。
- `--verbosity` オプションを使うには、そのオプションにひとつの値を指定しなければなりません。

上記の例では、`--verbosity` に任意の整数を取れます。この簡単なプログラムでは、実際には `True` または `False` の二つの値だけが有効です。そうなるようにコードを修正してみましょう:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
```

(次のページに続く)

```

        action="store_true")
args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")

```

実行してみましょう:

```

$ 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

```

こんなことが起こりました:

- オプションは値を必要とするのではなく、単なるフラグになりました。この考え方に沿うようにオプションの名前も変更しています。ここで新しいキーワード `action` を指定し、値として `"store_true"` を設定していることに注意してください。これは、オプションが指定された場合に値として `True` を `args.verbose` に設定するということを意味します。オプションの指定がない場合の値は `False` となることを意味します。
- フラグは値を取るべきではないので、値を指定するとプログラムはエラーになります。
- ヘルプテキストが変わっています。

## 4.1 短いオプション

コマンドラインになれば、オプションの短いバージョンの話題に触れていないことに気付いたでしょう。それはとても簡単です:

```

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")

```

上記のプログラムを実行するとこうなります:

```
$ 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
```

新しい機能がヘルプテキストにも反映されている点に気付いたでしょう。

## 5 位置引数と Optional 引数の併用

プログラムが複雑になってきました:

```
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)
```

出力は以下のようになります:

```
$ 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
```

- 位置引数を元に戻したので、引数を指定しないとエラーになりました。
- 2つのオプションの順序を考慮しないことに注意してください。

複数の詳細レベルを値にとれるようにプログラムを元に戻して、指定された値を使ってみましょう:



```

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)

```

実行してみましょう:

```

$ 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

```

プログラムのバグである最後の結果を除いて、上手く行っているようです。このバグを、`--verbosity` オプションが取れる値を制限することで修正しましょう:

```

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:

```

(次のページに続く)

```

    print(f"{args.square}^2 == {answer}")
else:
    print(answer)

```

実行してみましょう:

```

$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

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

```

変更がエラーメッセージとヘルプメッセージの両方に反映されていることに注意してください。

詳細レベルついて、違ったアプローチを試してみましょう。これは CPython 実行可能ファイルがその詳細レベル引数を扱う方法と同じです。( `python --help` の出力を確認してください) :

```

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)

```

もう一つの action である "count" を紹介します。これは指定されたオプションの出現回数を数えます:

```

$ 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

```

- 前のバージョンのスクリプトのようにフラグ (action="store\_true" に似ています) になりました。エラーとなる理由がわかります。(訳注: 5つ目の例では -v オプションに引数を与えたため、エラーとなっています。)
- これは "store\_true" アクションによく似た動作をします。
- では "count" アクションが何をもたらすかデモンストレーションをします。おそらくこのような使用方法を前に見たことがあるでしょう。
- -v フラグを指定しなければ、フラグの値は None とみなされます。
- 期待通り、長い形式のフラグを指定しても同じ結果になります。
- 残念ながら、このヘルプ出力はプログラムが新しく得た機能についてそこまで有益ではありません。しかし、スクリプトのドキュメンテーションを改善することでいつでも修正することができます (例えば、help キーワード引数を使用することで)。
- 最後の出力はプログラムにバグがあることを示します。

修正しましょう:

```

import argparse
parser = argparse.ArgumentParser()

```

(次のページに続く)

```

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)

```

これが結果です:

```

$ 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'

```

- 最初の出力は上手くいっていますし、以前のバグが修正されています。最も詳細な出力を得るには、2以上の値が必要です。
- 三番目の結果は、よくありません。

このバグを修正しましょう:

```

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:

```

(前のページからの続き)

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

もう一つのキーワード引数である `default` を導入しました。整数値と比較できるように、その値に 0 を設定しました。デフォルトでは、optional 引数が指定されていない場合 `None` となること、`None` が整数値と比較できない（よって `TypeError` 例外となる）ことを思い出してください。

こうなりました:

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

ここまで学んできたことだけで、さまざまな事が実現できます。しかしまだ表面をなぞっただけです。`argparse` モジュールはとても強力ですので、チュートリアルを終える前にもう少しだけ探検してみましょう。

## 6 もうちょっとだけ学ぶ

もし、この小さなプログラムを二乗以外の累乗が行えるように拡張するとどうなるでしょうか:

```
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)
```

出力:

```
$ 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
```

(次のページに続く)

```
positional arguments:
  x                  the base
  y                  the exponent

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

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

これまで、出力されるテキストを **変更する** ために詳細レベルを使ってきました。かわりに下記の例では、**追加の** テキストを出力するのに詳細レベルを使用します:

```
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)
```

出力:

```
$ 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 多義性のある引数の指定

ある引数の解釈について、位置引数なのか、ある引数に対する追加の引数なのか、あいまいさがあるときは、`--` を使うことで `parse_args()` に対してこれ以降は位置引数であることを伝えることができます:

```

>>> 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 競合するオプション

これまでのところ、`argparse.ArgumentParser` インスタンスの2つのメソッドについて学んできました。ここで3つめのメソッドとして `add_mutually_exclusive_group()` を導入しましょう。このメソッドは互いに対立する引数の指定を可能にします。また、新しい機能として理解しやすいように、プログラムの残りの部分を変更しましょう: 以下では `--quiet` オプションを `--verbose` と反対のオプションとして導入します:

```

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}")

```

プログラムはより簡潔になりましたが、デモのための機能が失われました。ともかく下記が実行結果です:

```
$ 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
```

これは分かりやすいでしょう。ここで得たちょっとした柔軟性を示すために最後の出力を追加しました、つまり長い形式のオプションと短い形式のオプションの混在です。

結びの前に、恐らくあなたはプログラムの主な目的をユーザに伝えたいでしょう。万が一、彼らがそれを知らないときに備えて:

```
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}")
```

使用方法のテキストが少し変化しました。[-v | -q] は -v または -q のどちらかを使用できるが、同時に両方を使用できないことを意味します:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y
```

(次のページに続く)



```

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 argparse の出力を翻訳するには

ヘルプテキストやエラーメッセージなどの `argparse` モジュールの出力は、すべて `gettext` モジュールを使って訳せるように作られています。これにより、`argparse` によって生成されたメッセージを簡単にローカライズできます。il8n-howto も参照してください。

たとえば、この `argparse` の出力では:

```

$ 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

```

文字列 `usage:`, `positional arguments:`, `options:` と `show this help message and exit` はすべて訳せます。

これらの文字列を訳すには、それらをまず `.po` ファイルに抽出する必要があります。たとえば、[Babel](#) を使い、次のコマンドを実行します。

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

このコマンドは `argparse` モジュールからすべての翻訳可能な文字列を抽出し、それらを `messages.po` という名前のファイルに出力します。このコマンドは Python のインストールが `/usr/lib` にあることを前提と

しています。

システム上の `argparse` モジュールの場所は、次のスクリプトで調べられます:

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

.po ファイル内のメッセージが訳され、その訳が `gettext` を用いてインストールされると、`argparse` は翻訳されたメッセージを表示できるようになります。

`argparse` 出力内のあなたの文字列を翻訳するには、`gettext` を使います。

## 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)
```

出力:

```
$ 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 結び

`argparse` モジュールはここで学んだことより多くの機能を提供します。モジュールのドキュメントはとても詳細で、綿密で、そしてたくさんの例があります。このチュートリアルを体験したことで、気がめいることなくそれらの他の機能を会得できるに違いありません。