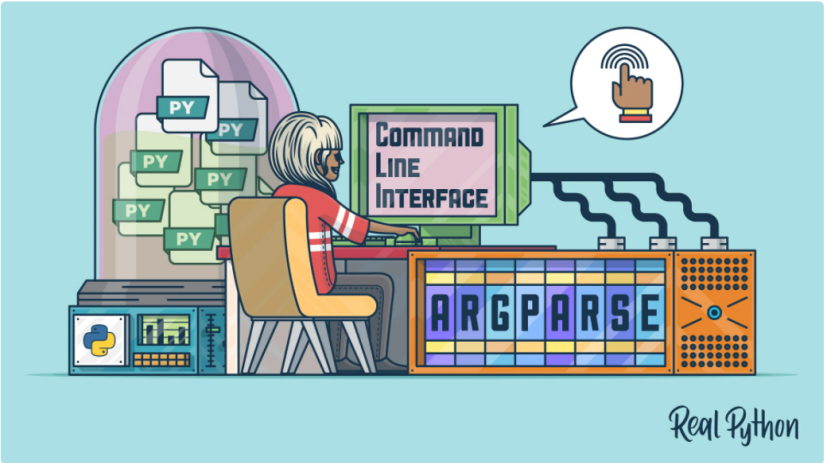

How to Build Command Line Interfaces in Python With argparse

by Davide Mastromatteo Jun 12, 2019 17 Comments best-practices intermediate

Mark as Completed

Tweet Share Email

Table of Contents

One of the strengths of Python is that it comes with batteries included: it has a rich and versatile standard library that makes it one of the best programming languages for writing scripts for the command line. But, if you write scripts for the command line, then you also need to provide a good command line interface, which you can create with the Python `argparse` library.

Table of Contents

- What Is a Command Line Interface?
- When to Use a Command Line Interface
- How to Use the Python argparse Library to Create a Command Line Interface
- The Advanced Use of the Python argparse Library
- Conclusion

Mark as Completed

Tweet Share Email

In this article, you'll learn:

- What the Python `argparse` library is, and why it's important to use it if you need to write command line scripts in Python
- How to use the Python `argparse` library to quickly create a simple CLI in Python
- What the advanced usage of the Python `argparse` library is

This article is written for early intermediate Pythonistas who probably write scripts in Python for their everyday work but have never implemented a command line interface for their scripts.

If that sounds like you, and you're used to setting variable values at the beginning of your scripts or manually parsing the `sys.argv` system list instead of using a more robust CLI development tool, then this article is for you.

What Is a Command Line Interface?

The command line interface (also known as **CLI**) is a means to interact with a command line script. Python comes with several different libraries that allow you to write a command line interface for your scripts, but the standard way for creating a CLI in Python is currently the Python `argparse` library.

The Python `argparse` library was released as part of the standard library with Python 3.2 on February the 20th, 2011. It was introduced with Python Enhancement Proposal 389 and is now the standard way to create a CLI in Python, both in 2.7 and 3.2+ versions.

This new module was released as a replacement for the older `getopt` and `optparse` modules because they were lacking some important features.

The Python `argparse` library:

- Allows the use of positional arguments
- Allows the customization of the prefix chars
- Supports variable numbers of parameters for a single option
- Supports subcommands (A main command line parser can use other command line parsers depending on some arguments.)

Before getting started, you need to know how a command line interface works, so open a terminal on your computer and execute the command `ls` to get the list of the files contained in the current directory like this:

Shell

```
$ ls
dcdb_20180201.sg4    mastro35.sg4        openings.sg4
dcdb_20180201.si4    mastro35.si4        openings.si4
dcdb_20180201.sn4    mastro35.sn4        openings.sn4
```

As you can see, there are a bunch of files in the current directory, but the command didn't return a lot of information about these files.

The good news is that you don't need to look around for another program to have a richer list of the files contained in the current directory. You also don't need to modify the `ls` command yourself, because it adopts a command line interface, that is just a set of tokens (called **arguments**) that you can use to configure the behavior of this command.

Now try to execute the command `ls` again, but with adding the `-l` option to the command line as in the example below:

Shell

```
$ ls -l
```

```
total 641824
-rw-------  1 dave  staff  204558286  5 Mar  2018 dcdb_20180201.sg4
-rw-------  1 dave  staff  110588409  5 Mar  2018 dcdb_20180201.si4
-rw-------  1 dave  staff    2937516  5 Mar  2018 dcdb_20180201.sn4
-rw-------  1 dave  staff     550127 27 Mar  2018 mastro35.sg4
-rw-------  1 dave  staff      15974 11 Gen 17:01 mastro35.si4
-rw-------  1 dave  staff       3636 27 Mar  2018 mastro35.sn4
-rw-------  1 dave  staff      29128 17 Apr  2018 openings.sg4
-rw-------  1 dave  staff        276 17 Apr  2018 openings.si4
-rw-------  1 dave  staff         86 18 Apr  2018 openings.sn4
```

The output is very different now. The command has returned a lot of information about the permissions, owner, group, and size of each file and the total directory occupation on the disk.

This is because you used the command line interface of the `ls` command and specified the `-l` option that enables the long format, a special format that returns a lot more information for every single file listed.

In order to familiarize yourself with this topic, you're going to read a lot about arguments, options, and parameters, so let's clarify the terminology right away:

- An **argument** is a single part of a command line, delimited by blanks.
- An **option** is a particular type of argument (or a part of an argument) that can modify the behavior of the command line.
- A **parameter** is a particular type of argument that provides additional information to a single option or command.

Consider the following command:

Shell

```
$ ls -l -s -k /var/log
```

In this example, you have five different arguments:

1. **`ls`:** the name of the command you are executing
2. **`-l`:** an option to enable the long list format
3. **`-s`:** an option to print the allocated size of each file
4. **`-k`:** an option to have the size in kilobytes
5. **`/var/log`:** a parameter that provides additional information (the path to list) to the command

Note that, if you have multiple options in a single command line, then you can combine them into a single argument like this:

Shell

```
$ ls -lsk /var/log
```

Here you have only three arguments:

1. **`ls` :** the name of the command you are executing
2. **`-lsk`:** the three different options you want to enable (a combination of `-l`, `-s`, and `-k`)
3. **`/var/log`:** a parameter that provides additional information (the path to list) to the command

When to Use a Command Line Interface

Now that you know what a command line interface is, you may be wondering when it's a good idea to implement one in your programs. The rule of thumb is that, if you want to provide a user-friendly approach to configuring your program, then you should consider a

command line interface, and the standard way to do it is by using the Python `argparse` library.

Even if you're creating a complex command line program that needs a configuration file to work, if you want to let your user specify which configuration file to use, it's a good idea to accept this value by creating a command line interface with the Python `argparse` library.

How to Use the Python `argparse` Library to Create a Command Line Interface

Using the Python `argparse` library has four steps:

1. Import the Python `argparse` library
2. Create the parser
3. Add optional and positional arguments to the parser
4. Execute `.parse_args()`

After you execute `.parse_args()`, what you get is a `Namespace` object that contains a simple property for each input argument received from the command line.

In order to see these four steps in detail with an example, let's pretend you're creating a program named `myls.py` that lists the files contained in the current directory. Here's a possible implementation of your command line interface without using the Python `argparse` library:

Python

```python
# myls.py
import os
import sys

if len(sys.argv) > 2:
    print('You have specified too many arguments')
    sys.exit()

if len(sys.argv) < 2:
    print('You need to specify the path to be listed')
    sys.exit()

input_path = sys.argv[1]

if not os.path.isdir(input_path):
    print('The path specified does not exist')
    sys.exit()

print('\n'.join(os.listdir(input_path)))
```

This is a possible implementation of the command line interface for your program that doesn't use the Python `argparse` library, but if you try to execute it, then you'll see that it works:

Shell

```
$ python myls.py
You need to specify the path to be listed

$ python myls.py /mnt /proc /dev
You have specified too many arguments

$ python myls.py /mnt
dir1
dir2
```

As you can see, the script does work, but the output is quite different from the output you'd expect from a standard built-in command.

Now, let's see how the Python `argparse` library can improve this code:

Python

```python
# myls.py
# Import the argparse library
```

```
# Import the argparse library
import argparse

import os
import sys

# Create the parser
my_parser = argparse.ArgumentParser(description='List the content of a folder')

# Add the arguments
my_parser.add_argument('Path',
                       metavar='path',
                       type=str,
                       help='the path to list')

# Execute the parse_args() method
args = my_parser.parse_args()

input_path = args.Path

if not os.path.isdir(input_path):
    print('The path specified does not exist')
    sys.exit()

print('\n'.join(os.listdir(input_path)))
```

The code has changed a lot with the introduction of the Python `argparse` library.

The first big difference compared to the previous version is that the `if` statements to check the arguments provided by the user are gone because the library will check the presence of the arguments for us.

We've imported the Python `argparse` library, created a simple parser with a brief description of the program's goal, and defined the positional argument we want to get from the user. Lastly, we have executed `.parse_args()` to parse the input arguments and get a `Namespace` object that contains the user input.

Now, if you run this code, you'll see that with just four lines of code. You have a very different output:

Shell

```
$ python myls.py
usage: myls.py [-h] path
myls.py: error: the following arguments are required: path
```

As you can see, the program has detected that you needed at least a positional argument (`path`), and so the execution of the program has been interrupted with a specific error message.

You may also have noticed that now your program accepts an optional `-h` flag, like in the example below:

Shell

```
$ python myls.py -h
usage: myls.py [-h] path

List the content of a folder

positional arguments:
path        the path to list

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

Good, now the program responds to the `-h` flag, displaying a help message that tells the user how to use the program. Isn't that neat, considering that you didn't even need to ask for that feature?

Lastly, with just four lines of code, now the `args` variable is a `Namespace` object, which has a property for each argument that has been gathered from the command line. That's super convenient.

The Advanced Use of the Python `argparse` Library

In the previous section, you learned the basic usage of the Python `argparse` library, and now you can implement a simple command line interfaces for all your programs. However, there's a lot more that you can achieve with this library. In this section, you'll see almost everything this library can offer you.

Setting the Name of the Program

By default, the library uses the value of the `sys.argv[0]` element to set the name of the program, which as you probably already know is the name of the Python script you have executed. However, you can specify the name of your program just by using the `prog` keyword:

Python

```
# Create the parser
my_parser = argparse.ArgumentParser(prog='myls',
                                    description='List the content of a folder')
```

With the `prog` keyword, you specify the name of the program that will be used in the help text:

Shell

```
$ python myls.py
usage: myls [-h] path
myls.py: error: the following arguments are required: path
```

As you can see, now the program name is just `myls` instead of `myls.py`.

Displaying a Custom Program Usage Help

By default, the program usage help has a standard format defined by the Python `argparse` library. However, you can customize it with the `usage` keyword like this:

Python

```
# Create the parser
my_parser = argparse.ArgumentParser(prog='myls',
                                    usage='%(prog)s [options] path',
                                    description='List the content of a folder')
```

Note that, at runtime, the `%(prog)s` token is automatically replaced with the name of your program:

Shell

```
$ python myls.py
usage: myls [options] path
myls: error: too few arguments
```

As you can see, the help of the program now shows a different usage string, where the `[-h]` option has been replaced by a generic `[options]` token.

Displaying Text Before and After the Arguments Help

To customize the text displayed before and after the arguments help text, you can use two different keywords:

1. **`description`:** for the text that is shown before the help text

2. **`epilog`:** for the text shown after the help text

You've already seen the `description` keyword in the previous chapter, so let's see an example of how the `epilog` keyword works:

Python

```
# Create the parser
my_parser = argparse.ArgumentParser(description='List the content of a folder',
                                    epilog='Enjoy the program! :)')
```

The `epilog` keyword here has customized the text that will be shown after the standard help text:

Shell

```
$ python myls.py -h
usage: myls.py [-h] path

List the content of a folder

positional arguments:
path        the path to list

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

Enjoy the program! :)
```

Now the output shows the extra text that has been customized by the `epilog` keyword.

Customizing the Allowed Prefix Chars

Another feature that the Python `argparse` library offers you is the ability to customize the **prefix chars**, which are the chars that you can use to pass optional arguments to the command line interface.

By default, the standard prefix char is the dash (`-`) character, but if you want to use a different character, then you can customize it by using the `prefix_chars` keyword while defining the parser like this:

Python

```
# Create the parser
my_parser = argparse.ArgumentParser(description='List the content of a folder',
                                    epilog='Enjoy the program! :)',
                                    prefix_chars='/')
```

After the redefinition, the program now supports a completely different prefix char, and the help text has changed accordingly:

Shell

```
$ python myls.py
usage: myls.py [/h] path
myls.py: error: too few arguments
```

As you can see, now your program does not support the `-h` flag but the `/h` flag. That's especially useful when you're coding for Microsoft Windows because Windows users are used to these prefix chars when working with the command line.

Setting Prefix Chars for Files That Contain Arguments to Be Included

When you are dealing with a very long or complicated command line, it can be a good idea

to save the arguments to an external file and ask your program to load arguments from it. The Python `argparse` library can do this work for you out of the box.

To test this feature, create the following Python program:

Python

```
# fromfile_example.py
import argparse

my_parser = argparse.ArgumentParser(fromfile_prefix_chars='@')

my_parser.add_argument('a',
                       help='a first argument')

my_parser.add_argument('b',
                       help='a second argument')

my_parser.add_argument('c',
                       help='a third argument')

my_parser.add_argument('d',
                       help='a fourth argument')

my_parser.add_argument('e',
                       help='a fifth argument')

my_parser.add_argument('-v',
                       '--verbose',
                       action='store_true',
                       help='an optional argument')

# Execute parse_args()
args = my_parser.parse_args()

print('If you read this line it means that you have provided '
      'all the parameters')
```

Note that we have used the `fromfile_prefix_chars` keyword while creating the parser.

Now, if you try to execute your program without passing any arguments, then you'll get an error message:

Shell

```
$ python fromfile_example.py
usage: fromfile_example.py [-h] [-v] a b c d e
fromfile_example.py: error: the following arguments are required: a, b, c, d, e
```

Here you can see that the Python `argparse` library is complaining because you have not provided enough arguments.

So let's create a file named `args.txt` that contains all the necessary parameters, with an argument on each line like this:

Text

```
first
second
third
fourth
fifth
```

Now that you have specified a prefix char to get arguments from an external file, open a terminal and try to execute the previous program:

Shell

```
$ python fromfile_example.py @args.txt
If you read this line it means that you have provided all the parameters
```

In this example, you can see that `argparse` has read the arguments from the `args.txt` file.

Allowing or Disallowing Abbreviations

One of the features that the Python `argparse` library provides out of the box is the ability to handle abbreviations. Consider the following program, which prints out the value you specify on the command line interface for the `--input` argument:

Python

```
# abbrev_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('--input', action='store', type=int, required=True)
my_parser.add_argument('--id', action='store', type=int)

args = my_parser.parse_args()

print(args.input)
```

This program prints out the value you specify for the `--input` argument. We haven't looked at the optional arguments yet, but don't worry, we will discuss them in depth in just a moment. For now, just consider this argument like any other positional argument we already saw, with the difference that the name starts with a couple of dashes.

Now let's see how the Python `argparse` library can handle abbreviations, by calling our program multiple times, specifying a different abbreviation of the `input` argument at each run:

Shell

```
$ python abbrev_example.py --input 42
42

$ python abbrev_example.py --inpu 42
42

$ python abbrev_example.py --inp 42
42

$ python abbrev_example.py --in 42
42
```

As you can see, the optional parameters can always be shortened unless the abbreviation can lead to an incorrect interpretation. But what happens if you try to execute the program specifying just `--i 42`? In this case, `argparse` doesn't know if you want to pass the value `42` to the `--input` argument or to the `--id` argument, so it exits the program with a specific error message:

Shell

```
$ python abbrev_example.py --i 42
usage: abbrev_example.py [-h] --input INPUT [--id ID]
abbrev_example.py: error: ambiguous option: --i could match --input, --id
```

However, if you don't like this behavior, and you want to force your users to specify the full name of the options they use, then you can just disable this feature with the keyword `allow_abbrev` set to `False` during the creation of the parser:

Python

```
# abbrev_example.py
import argparse

my_parser = argparse.ArgumentParser(allow_abbrev=False)
my_parser.add_argument('--input', action='store', type=int, required=True)

args = my_parser.parse_args()

print(args.input)
```

```
print(args.input)
```

Now, if you try the code above, you'll see that the abbreviations are no longer permitted:

Shell

```
$ python abbrev_example.py --inp 42
usage: abbrev_example.py [-h] --input INPUT
abbrev_example.py: error: the following arguments are required: --input
```

The error message tells the user that the `--input` parameter has not been specified because the `--inp` abbreviation has not been recognized.

Using Auto Help

In some of the previous examples, you used the `-h` flag to get a help text. This is a very convenient feature that the Python `argparse` library allows you to use without having to code anything. However, sometimes you may want to disable this feature. To do that, just use the `add_help` keyword when creating the parser:

Python

```
# Create the parser
my_parser = argparse.ArgumentParser(description='List the content of a folder',
                                    add_help=False)
```

The code in the example above specifies the `add_help` keyword set to `False`, so now if you run the code, you'll see that the `-h` flag isn't accepted anymore:

Shell

```
$ myls.py
usage: myls.py path
myls.py: error: the following arguments are required: path
```

As you can see, the `-h` flag is no longer shown or accepted.

Setting the Name or Flags of the Arguments

There are basically two different types of arguments that you can add to your command line interface:

1. Positional arguments
2. Optional arguments

Positional arguments are the ones your command needs to operate.

In the previous example, the argument `path` was a positional argument, and our program couldn't work without it. They are called **positional** because their position defines their function.

For example, consider the `cp` command on Linux (or the `copy` command in Windows). Here's the standard usage:

Shell

```
$ cp [OPTION]... [-T] SOURCE DEST
```

The first positional argument after the `cp` command is the source of the file you're going to copy. The second one is the destination where you want to copy it.

Optional arguments are not mandatory, and when they are used they can modify the behavior of the command at runtime. In the `cp` example, an optional argument is, for example, the `-r` flag, which makes the command copy directories recursively.

Syntactically, the difference between positional and optional arguments is that optional arguments start with `-` or `--`, while positional arguments don't.

To add an optional argument, you just need to call `.add_argument()` again and name the new argument with a starting `-`.

For example, try to modify the `myls.py` like this:

Python

```
# myls.py
# Import the argparse library
import argparse

import os
import sys

# Create the parser
my_parser = argparse.ArgumentParser(description='List the content of a folder')

# Add the arguments
my_parser.add_argument('Path',
                       metavar='path',
                       type=str,
                       help='the path to list')
my_parser.add_argument('-l',
                       '--long',
                       action='store_true',
                       help='enable the long listing format')

# Execute parse_args()
args = my_parser.parse_args()

input_path = args.Path

if not os.path.isdir(input_path):
    print('The path specified does not exist')
    sys.exit()

for line in os.listdir(input_path):
    if args.long:  # Simplified long listing
        size = os.stat(os.path.join(input_path, line)).st_size
        line = '%10d  %s' % (size, line)
    print(line)
```

Now, try to execute this program to see if the new `-l` option is accepted:

Shell

```
$ python myls.py -h
usage: myls.py [-h] [-l] path

List the content of a folder

positional arguments:
path        the path to list

optional arguments:
-h, --help  show this help message and exit
-l, --long  enable the long listing format
```

As you can see, now the program also accepts (but doesn't require) the `-l` option, which allows the user to get a long listing format for the directory content.

Setting the Action to Be Taken for an Argument

When you add an optional argument to your command line interface, you can also define what kind of action to take when the argument is specified. This means that you usually need to specify how to store the value to the `Namespace` object you will get when `.parse_args()` is executed.

There are several actions that are already defined and ready to be used. Let's analyze them in detail:

- **`store`** stores the input value to the `Namespace` object. (This is the default action.)
- **`store_const`** stores a constant value when the corresponding optional arguments are specified.
- **`store_true`** stores the Boolean value `True` when the corresponding optional argument is specified and stores a `False` elsewhere.
- **`store_false`** stores the Boolean value `False` when the corresponding optional argument is specified and stores `True` elsewhere.
- **`append`** stores a list, appending a value to the list each time the option is provided.
- **`append_const`** stores a list appending a constant value to the list each time the option is provided.
- **`count`** stores an `int` that is equal to the times the option has been provided.
- **`help`** shows a help text and exits.
- **`version`** shows the version of the program and exits.

Let's create an example to test all the actions we have seen so far:

Python

```python
# actions_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.version = '1.0'
my_parser.add_argument('-a', action='store')
my_parser.add_argument('-b', action='store_const', const=42)
my_parser.add_argument('-c', action='store_true')
my_parser.add_argument('-d', action='store_false')
my_parser.add_argument('-e', action='append')
my_parser.add_argument('-f', action='append_const', const=42)
my_parser.add_argument('-g', action='count')
my_parser.add_argument('-i', action='help')
my_parser.add_argument('-j', action='version')

args = my_parser.parse_args()

print(vars(args))
```

This script accepts an optional argument for each type of action discussed and then prints the value of the arguments read from the command line. Test it by executing this example:

Shell

```
$ python actions_example.py
{'a': None, 'b': None, 'c': False, 'd': True, 'e': None, 'f': None, 'g': None}
```

As you can see, if we do not specify any arguments, then the default values are generally `None`, at least for the actions that do not store a Boolean value.

The use of the `store` action, instead, stores the value we pass without any further consideration:

Shell

```
$ python actions_example.py -a 42
{'a': '42', 'b': None, 'c': False, 'd': True, 'e': None, 'f': None, 'g': None}

$ python actions_example.py -a "test"
{'a': 'test', 'b': None, 'c': False, 'd': True, 'e': None, 'f': None, 'g': None}
```

The `store_const` action, stores the defined const when the arguments are provided. In our test, we provided just the `b` argument and the value of `args.b` is now `42`:

Shell

```
$ python actions_example.py -b
{'a': None, 'b': 42, 'c': False, 'd': True, 'e': None, 'f': None, 'g': None}
```

The `store_true` action stores a `True` Boolean when the argument is passed and store a `False` Boolean elsewhere. If you need the opposite behavior, just use the `store_false` action:

Shell

```
$ python actions_example.py
{'a': None, 'b': None, 'c': False, 'd': True, 'e': None, 'f': None, 'g': None}
$ python actions_example.py -c
{'a': None, 'b': None, 'c': True, 'd': True, 'e': None, 'f': None, 'g': None}
$ python actions_example.py -d
{'a': None, 'b': None, 'c': False, 'd': False, 'e': None, 'f': None, 'g': None}
```

The `append` action lets you create a list of all the values passed to the CLI with the same argument:

Shell

```
$ python actions_example.py -e me -e you -e us
{'a': None, 'b': None, 'c': False, 'd': True, 'e': ['me', 'you', 'us'], 'f': None, 'g
```

The `append_const` action is similar to the `append` one, but it always appends the same constant value:

Shell

```
$ python actions_example.py -f -f
{'a': None, 'b': None, 'c': False, 'd': True, 'e': None, 'f': [42, 42], 'g': None}
```

The `count` action counts how many time an argument is passed. It's quite useful when you want to implement a verbosity level for your program, since you can define a level where `-v` is less verbose than `-vvv`:

Shell

```
$ python actions_example.py -ggg
{'a': None, 'b': None, 'c': False, 'd': True, 'e': None, 'f': None, 'g': 3}
$ python actions_example.py -ggggg
{'a': None, 'b': None, 'c': False, 'd': True, 'e': None, 'f': None, 'g': 5}
```

The `help` action is the one you already saw at the beginning of the article. It's enabled for the `-h` flag by default, but you can use it for another flag if you want:

Shell

```
$ python actions_example.py -i
usage: actions_example.py [-h] [-a A] [-b] [-c] [-d] [-e E] [-f] [-g] [-i]
[-j]

optional arguments:
-h, --help  show this help message and exit
-a A
-b
-c
-d
-e E
-f
-g
-i
-j          show program's version number and exit
```

The `version` action is the last one you can use. It just shows the version of the program (defined by assigning a value to the `.version` property of the parser) and then ends the execution of the script:

Shell

```
$ python actions_example.py -j
1.0
```

Another possibility you have is to create a custom action. That's done by subclassing the `argparse.Action` class and implementing a couple of methods.

Look at the following example, which is a custom `store` action that is just a little bit more verbose than the standard one:

Python

```python
# custom_action.py
import argparse

class VerboseStore(argparse.Action):
    def __init__(self, option_strings, dest, nargs=None, **kwargs):
        if nargs is not None:
            raise ValueError('nargs not allowed')
        super(VerboseStore, self).__init__(option_strings, dest, **kwargs)

    def __call__(self, parser, namespace, values, option_string=None):
        print('Here I am, setting the ' \
              'values %r for the %r option...' % (values, option_string))
        setattr(namespace, self.dest, values)

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-i', '--input', action=VerboseStore, type=int)

args = my_parser.parse_args()

print(vars(args))
```

This example defines a custom action that is just like the `store` action but a little bit more verbose. Try to execute it to test how it works:

Shell

```
$ python custom_action.py -i 42
Here I am, setting the values 42 for the '-i' option...
{'input': 42}
```

As you can see, the program has printed out a line before setting the value `42` for the `-i` parameter.

Setting the Number of Values That Should Be Consumed by the Option

The parser, by default, assumes that you'll consume a single parameter for each argument, but you can modify this default behavior by specifying a different number of values with the `nargs` keyword.

For example, if you want to create an optional argument that consumes exactly three values, then you can specify the number `3` as the value for the `nargs` keyword while adding the parameter to the parser:

Python

```python
# nargs_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('--input', action='store', type=int, nargs=3)

args = my_parser.parse_args()

print(args.input)
```

Now, the program accepts three values for the `--input` parameter:

Shell

```
$ python nargs_example.py --input 42
usage: nargs_example.py [-h] [--input INPUT INPUT INPUT]
nargs_example.py: error: argument --input: expected 3 arguments

$ python nargs_example.py --input 42 42 42
[42, 42, 42]
```

As you can see, the value of the `args.input` variable is now a list that contains three values.

However, the `nargs` keyword can also accept the following:

- **`?`:** a single value, which can be optional
- **`*`:** a flexible number of values, which will be gathered into a list
- **`+`:** like `*`, but requiring *at least* one value
- **`argparse.REMAINDER`:** all the values that are remaining in the command line

So, for example, in the following program, the positional argument `input` takes a single value when provided, but if the value is not provided, then the one specified by the `default` keyword is used:

Python

```python
# nargs_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('input',
                       action='store',
                       nargs='?',
                       default='my default value')

args = my_parser.parse_args()

print(args.input)
```

Now you can choose to set a specific value for the `input` argument or not. In this case, the default value will be used:

Shell

```
$ python nargs_example.py 'my custom value'
my custom value

$ python nargs_example.py
my default value
```

To take a flexible number of values and gather them all into a single list, you need to specify the `*` value for the `nargs` keyword like this:

Python

```python
# nargs_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('input',
                       action='store',
                       nargs='*',
                       default='my default value')

args = my_parser.parse_args()

print(args.input)
```

See how this code allows the user to set a flexible number of values for the expected argument:

Shell

```
$ python nargs_example.py me you us
['me', 'you', 'us']

$ python nargs_example.py
my default value
```

If you need to take a variable number of values, but you have to be sure that at least one value is specified, then you can use the `+` value for the `nargs` keyword like this:

Python

```python
# nargs_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('input', action='store', nargs='+')

args = my_parser.parse_args()

print(args.input)
```

In this case, if you execute the program with no positional arguments, then you will receive an explicit error message:

Shell

```
$ python nargs_example.py me you us
['me', 'you', 'us']

$ python nargs_example.py
usage: nargs_example.py [-h] input [input ...]
nargs_example.py: error: the following arguments are required: input
```

Lastly, if you need to grab all the remaining arguments that have been specified on the command line and put them all in a list, then the `nargs` keyword has to be set to `argparse.REMAINDER` like this:

Python

```python
# nargs_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('first', action='store')
my_parser.add_argument('others', action='store', nargs=argparse.REMAINDER)

args = my_parser.parse_args()

print('first = %r' % args.first)
print('others = %r' % args.others)
```

Now if you execute this program, you will see that the first value will be associated with the first parameters, while all the other values provided will be associated with the second one:

Shell

```
$ python nargs_example.py me you us
first = 'me'
others = ['you', 'us']
```

Note how all the remaining values are put in a single list.

Setting a Default Value Produced if the Argument Is Missing

You already know that the user can decide whether or not to specify optional arguments in the command line. When arguments are not specified, the corresponding value is generally set to `None`.

However, it is possible to define a default value for an argument when it's not provided:

Python

```python
# default_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-a', action='store', default='42')

args = my_parser.parse_args()
```

```
print(vars(args))
```

If you execute this example without passing the `-a` option, then this is the output you get:

Shell

```
$ python default_example.py
{'a': '42'}
```

You can see that now the option `-a` is set to `42`, even if you didn't explicitly set the value on the command line.

Setting the Type of the Argument

By default, all the input argument values are treated as if they were strings. However, it's possible to define the type for the corresponding property of the `Namespace` object you get after `.parse_args()` is invoked just by defining it with the `type` keyword like this:

Python

```
# type_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-a', action='store', type=int)

args = my_parser.parse_args()

print(vars(args))
```

Specifying the int value for the argument, you are telling `argparse` that the `.a` property of your `Namespace` object has to be an `int` (instead of a string):

Shell

```
$ python type_example.py -a 42
{'a': 42}
```

Besides, now the value of the argument is checked at runtime, and if there's a problem with the type of the value provided at the command line, then the execution is interrupted with a clear error message:

Shell

```
$ python type_example.py -a "that's a string"
usage: type_example.py [-h] [-a A]
type_example.py: error: argument -a: invalid int value: "that's a string"
```

In this case, the error message is very clear because it states that you were expected to pass an `int` instead of a string.

Setting a Domain of Allowed Values for a Specific Argument

Another interesting possibility with the Python `argparse` library creating a domain of allowed values for specific arguments. You can do this by providing a list of accepted values while adding the new option:

Python

```
# choices_ex.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-a', action='store', choices=['head', 'tail'])

args = my_parser.parse_args()
```

Please note that if you are accepting numeric values, then you can even use `range()` to

specify a range of accepted values:

Python

```
# choices_ex.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-a', action='store', type=int, choices=range(1, 5))

args = my_parser.parse_args()

print(vars(args))
```

In this case, the value provided on the command line will be automatically checked against the range defined:

Shell

```
$ python choices_ex.py -a 4
{'a': 4}

$ python choices_ex.py -a 40
usage: choices_ex.py [-h] [-a {1,2,3,4}]
choices_ex.py: error: argument -a: invalid choice: 40 (choose from 1, 2, 3, 4)
```

If the input number is outside the defined range, then you'll get an error message.

Setting Whether the Argument Is Required

If you want to force your user to specify the value for an optional argument, then you can use the `required` keyword:

Python

```
# required_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-a',
                       action='store',
                       choices=['head', 'tail'],
                       required=True)

args = my_parser.parse_args()

print(vars(args))
```

If you use the `required` keyword set to `True` for an optional argument, then the user will be forced to set a value for that argument:

Shell

```
$ python required_example.py
usage: required_example.py [-h] -a {head,tail}
required_example.py: error: the following arguments are required: -a

$ python required_example.py -a head
{'a': 'head'}
```

That said, please bear in mind that requiring an optional argument is usually considered bad practice since the user wouldn't expect to have to set a value for an argument that should be optional.

Showing a Brief Description of What an Argument Does

A great feature of the Python `argparse` library is that, by default, you have the ability to ask

for help just by adding the `-h` flag to your command line.

To make it even better, you can add help text to your arguments, so as to give the users even more help when they execute your program with the `-h` flag:

Python

```python
# help_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-a',
                       action='store',
                       choices=['head', 'tail'],
                       help='set the user choice to head or tail')

args = my_parser.parse_args()

print(vars(args))
```

This example shows you how to define a custom help text for the `-a` argument, and now the help text will be more clear for the user:

Shell

```
$ python help_example.py -h
usage: help_example.py [-h] [-a {head,tail}]

optional arguments:
-h, --help      show this help message and exit
-a {head, tail}  set the user choice to head or tail
```

Defining a help text for all the arguments is a really good idea because it makes the usage of your program more clear to the user.

Defining Mutually Exclusive Groups

Another interesting option you have when working with the Python `argparse` library is the ability to create a mutually exclusive group for options that cannot coexist in the same command line:

Python

```python
# groups.py
import argparse

my_parser = argparse.ArgumentParser()
my_group = my_parser.add_mutually_exclusive_group(required=True)

my_group.add_argument('-v', '--verbose', action='store_true')
my_group.add_argument('-s', '--silent', action='store_true')

args = my_parser.parse_args()

print(vars(args))
```

You can specify the `-v` or the `-s` flags, unless they aren't on the same command line, and also the help text that `argparse` provides reflects this constraint:

Shell

```
$ python groups.py -h
usage: groups.py [-h] (-v | -s)

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose
  -s, --silent

$ python groups.py -v -s
usage: groups.py [-h] (-v | -s)
groups.py: error: argument -s/--silent: not allowed with argument -v/--verbose
```

If you specify all the options of a mutually exclusive group on the same command line you

will get an error.

Setting the Argument Name in Usage Messages

If an argument accepts an input value, it can be useful to give this value a name that the parser can use to generate the help message, and this can be done by using the `metavar` keyword. In the following example, you can see how you can use the `metavar` keyword to specify a name for the value of the `-v` flag:

Python

```
# metavar_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-v',
                       '--verbosity',
                       action='store',
                       type=int,
                       metavar='LEVEL')

args = my_parser.parse_args()

print(vars(args))
```

Now, if you run your program with the `-h` flag, the help text assigns the name `LEVEL` to the value of the `-v` flag:

Shell

```
$ python metavar_example.py -h
usage: metavar_example.py [-h] [-v LEVEL]

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

Please note that, in the help message, the value accepted for the `-v` flag is now named `LEVEL`.

Setting the Name of the Attribute to Be Added to the Object Once Parsed

As you have already seen, when you add an argument to the parser, the value of this argument is stored in a property of the `Namespace` object. This property is named by default as the first argument passed to `.add_argument()` for the positional argument and as the long option string for optional arguments.

If an option uses dashes (as is fairly common), they will be converted to underscores in the property name:

Python

```
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-v',
                       '--verbosity-level',
                       action='store',
                       type=int)

args = my_parser.parse_args()
print(args.verbosity_level)
```

However, it's possible to specify the name of this property just by using the keyword `dest` when you're adding an argument to the parser:

Python

```
# dest_example.py
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-v',
                       '--verbosity',
                       action='store',
                       type=int,
                       dest='my_verbosity_level')

args = my_parser.parse_args()

print(vars(args))
```

By running this program, you'll see that now the `args` variable contains a `.my_verbosity_level` property, even if by default the name of the property should have been `.verbosity`:

Shell

```
$ python dest_example.py -v 42
{'my_verbosity_level': 42}
```

The default name of this property would have been `.verbosity`, but since a different name has been specified by the `dest` keyword, `.my_verbosity_level` has been used.

Conclusion

Now you know what a command line interface is and how you can create one in Python by using the Python `argparse` library.

In this article, you've learned:

- What the Python `argparse` library is, and why it's important to use it if you need to write command line scripts in Python
- How to use the Python `argparse` library to quickly create a simple CLI in Python
- What the advanced usage of the Python `argparse` library is

Writing a good command line interface is a good way to create self-explanatory programs and give users a means of interacting with your application.

If you still have questions, don't hesitate to reach out in the comment section below and take a look at the official documentation and the Tshepang Lekhonkhobe tutorial that is part of the official Python 3 HOWTO documentation.

About **Davide Mastromatteo**

Developer and editor of “the Python Corner". Blood donor, Apple user, Python and Swift addicted. NFL, Rugby and Chess lover. Constantly hungry and foolish.

» More about Davide

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Alex

 Aldren

 Geir Arne

 Joanna

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won’t make the cut here.

What’s your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate