

# Build Command-Line Interfaces With Python's argparse

by Leodanis Pozo Ramos · Dec 07, 2022 · 20 Comments · [intermediate](#) [python](#)

Mark as Completed



[Tweet](#)

[Share](#)

[Email](#)

## Table of Contents

- Getting to Know Command-Line Interfaces
  - Command-Line Interfaces (CLIs)
  - Commands, Arguments, Options, Parameters, and Subcommands

— FREE Email Series —



Python Tricks 

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

[Get Python Tricks »](#)

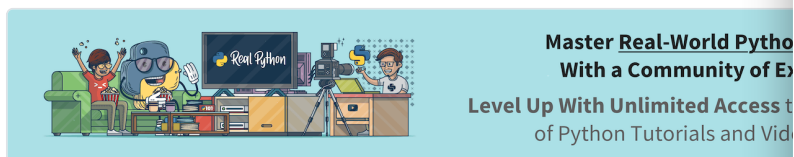


No spam. Unsubscribe any time.

## All Tutorial Topics

[advanced](#)
[api](#)
[basics](#)
[best-practices](#)
[community](#)
[databases](#)
[data-science](#)
[devops](#)
[django](#)
[docker](#)
[flask](#)
[front-end](#)
[gamedev](#)
[gui](#)
[intermediate](#)
[machine-learning](#)
[projects](#)
[python](#)
[testing](#)
[tools](#)
[web-dev](#)
[web-scraping](#)

- Getting Started With CLIs in Python: sys.argv vs argparse
  - Using sys.argv to Build a Minimal CLI
  - Creating a CLI With argparse
- Creating Command-Line Interfaces With Python's argparse
  - Creating a Command-Line Argument Parser
  - Adding Arguments and Options
  - Parsing Command-Line Arguments and Options
  - Setting Up Your CLI App's Layout and Build System
- Customizing Your Command-Line Argument Parser
  - Tweaking the Program's Help and Usage Conf
  - Providing Global Settings for Arguments and O
- Fine-Tuning Your Command-Line Arguments and Options
  - Setting the Action Behind an Option
  - Customizing Input Values in Arguments and Options
  - Providing and Customizing Help Messages in
- Defining Mutually Exclusive Argument and Option Groups
- Adding Subcommands to Your CLIs
- Handling How Your CLI App's Execution Terminates
- Conclusion



Master Real-World Python  
With a Community of Experts

Level Up With Unlimited Access to  
of Python Tutorials and Videos

Remove ads

**Watch Now** This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Building Command Line Interfaces With argparse**



Master Real-World Python Skills

## Improve Your Python

...with a fresh **Python Trick**   
code snippet every couple of days:

☐ Receive the Real Python newsletter and get notified about new tutorials we publish on the site, as well as occasional special offers.

**Send Python Tricks »**

- Adding Subcommands to Your CLIs
- Handling How Your CLI App's Execution Terminates
- Conclusion

Improve Your Python



Mark as Completed



Command-line apps may not be common in the general user's space, but they're present in

development, data science, systems administration, and many other operations. Every command-line app needs a user-friendly [command-line interface \(CLI\)](#) so that you can

interact with the app itself. In Python, you can create full-featured CLIs with the **argparse** module from the [standard library](#).

### In this article, you'll learn how to:

- Get started with **command-line interfaces**
- **Organize** and **lay out** a command-line app project in Python
- Create **command-line interfaces** with Python's **argparse**
- Deeply customize your CLIs with some **powerful features** of **argparse**

To get the most out of this tutorial, you should be familiar with Python programming, including concepts such as [object-oriented programming](#), [script development and execution](#), and Python [packages and modules](#). It'll also be helpful if you're familiar with general concepts and topics related to using a command line or terminal.

**Source Code:** Click [here](#) to download the source code that you'll use to build command-line interfaces with **argparse**.

## Getting to Know Command-Line Interfaces

Since the invention of computers, humans have always needed and found ways to interact and share information with these machines. The information exchange has flowed among humans, [computer software](#), and [hardware components](#). The shared boundary between any two of these elements is generically known as an [interface](#).

In software development, an interface is a special part of a given piece of software that allows interaction between components of a computer system. When it comes to human and software interaction, this vital component is known as the [user interface](#).

[Tweet](#)[Share](#)[Email](#)

### Recommended Video Course

[Building Command Line Interfaces With argparse](#)



Join Real Python and Unlock Learning Paths, Courses, Live Q&As, and More:

[Become a Python Expert »](#)

You'll find different types of user interfaces in programming. Probably, [graphical user interfaces \(GUIs\)](#) are the most common today. However, you'll also find apps and programs that provide [command-line interfaces \(CLIs\)](#) for their users. In this tutorial, you'll learn about CLIs and how to create them in Python.



Remove ads

## Command-Line Interfaces (CLIs)

**Command-line interfaces** allow you to interact with an application or program through your operating system command line, terminal, or console.

To understand command-line interfaces and how they work, consider this practical example. Say that you have a directory called `sample` containing three sample files. If you're on a [Unix-like](#) operating system, such as Linux or macOS, go ahead and open a command-line window or terminal in the parent directory and then execute the following command:

### Shell

```
$ ls sample/  
hello.txt      lorem.md      realpython.md
```

The `ls` [Unix command](#) lists the files and subdirectories contained in a target directory, which defaults to the current working directory. The above command call doesn't display much information about the content of `sample`. It only displays the filenames on the screen.

**Note:** If you're on Windows, then you'll have an `ls` command that works similarly to the Unix `ls` command. However, in its plain form, the command displays a different output.

#### Windows PowerShell

```
PS> ls .\sample\
```

Directory: C:\sample

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a--	11/10/2022 10:06 AM	88	hello.txt
-a--	11/10/2022 10:06 AM	2629	lorem.md
-a--	11/10/2022 10:06 AM	429	realpython.md

The PowerShell `ls` command issues a table containing detailed information on every file and subdirectory under your target directory. So, the upcoming examples won't work as expected on Windows systems.

Suppose you want richer information about your directory and its content. In that case, you don't need to look around for a program other than `ls` because this command has a full-featured command-line interface with a useful set of **options** that you can use to customize the command's behavior.

For example, go ahead and execute `ls` with the `-l` option:

#### Shell

```
$ ls -l sample/
total 24
-rw-r--r--@ 1 user  staff   83 Aug 17 22:15 hello.txt
-rw-r--r--@ 1 user  staff 2609 Aug 17 22:15 lorem.md
-rw-r--r--@ 1 user  staff  428 Aug 17 22:15 realpython.md
```

The output of `ls` is quite different now. The command displays much more information about the files in `sample`, including permissions, owner, group, date, and size. It also shows

about the files in sample, including permissions, owner, group, date, and size. It also shows the total space that these files use on your computer's disk.

**Note:** To get a detailed list of all the options that `ls` provides as part of its CLI, go ahead and run the `man ls` command in your command line or terminal.

This richer output results from using the `-l` option, which is part of the Unix `ls` command-line interface and enables the detailed output format.

## Commands, Arguments, Options, Parameters, and Subcommands

Throughout this tutorial, you'll learn about **commands** and **subcommands**. You'll also learn about command-line **arguments**, **options**, and **parameters**, so you should incorporate these terms into your tech vocabulary:

- **Command:** A program or routine that runs at the command line or terminal window. You'll typically identify a command with the name of the underlying program or routine.
- **Argument:** A required or optional piece of information that a command uses to perform its intended action. Commands typically accept one or many arguments, which you can provide as a whitespace-separated or comma-separated list on your command line.
- **Option**, also known as **flag** or **switch**: An optional argument that modifies a command's behavior. Options are passed to commands using a specific name, like `-l` in the previous example.
- **Parameter:** An argument that an option uses to perform its intended operation or action.

- **Subcommand:** A predefined name that can be passed to an application to run a specific action.

Consider the sample command construct from the previous section:

Shell

```
$ ls -l sample/
```

In this example, you've combined the following components of a CLI:

- **ls:** The command's name or the app's name
- **-l:** An option, switch, or flag that enables detailed outputs
- **sample:** An argument that provides additional information to the command's execution

Now consider the following command construct, which showcases the CLI of Python's package manager, known as [pip](#):

Shell

```
$ pip install -r requirements.txt
```

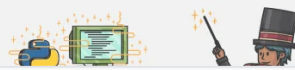
This is a common pip command construct, which you've probably seen before. It allows you to install the requirements of a given Python project using a `requirements.txt` file. In this example, you're using the following CLI components:


- **pip:** The command's name
- **install:** The name of a subcommand of pip
- **-r:** An option of the `install` subcommand
- **requirements.txt:** An argument, specifically a parameter of the `-r` option

Now you know what command-line interfaces are and what their main parts or components are. It's time to learn how to create your own CLI in Python.

are. It's time to learn how to create your own CLIs in Python.

**Improve Your Python with Python Tricks**



 Remove ads

## Getting Started With CLIs in Python: `sys.argv` vs `argparse`

Python comes with a couple of tools that you can use to write command-line interfaces for your programs and apps. If you need to quickly create a minimal CLI for a small program, then you can use the `argv` attribute from the `sys` module. This attribute automatically stores the arguments that you pass to a given program at the command line.

### Using `sys.argv` to Build a Minimal CLI

As an example of using `argv` to create a minimal CLI, say that you need to write a small program that lists all the files in a given directory, similar to what `ls` does. In this situation, you can write something like this:



## Python

```
# ls_argv.py

import sys
from pathlib import Path

if (args_count := len(sys.argv)) > 2:
    print(f"One argument expected, got {args_count - 1}")
    raise SystemExit(2)
elif args_count < 2:
    print("You must specify the target directory")
    raise SystemExit(2)

target_dir = Path(sys.argv[1])

if not target_dir.is_dir():
    print("The target directory doesn't exist")
    raise SystemExit(1)

for entry in target_dir.iterdir():
    print(entry.name)
```

This program implements a minimal CLI by manually processing the arguments provided at the command line, which are automatically stored in `sys.argv`. The first item in `sys.argv` is always the program's name. The second item will be the target directory. The app shouldn't accept more than one target directory, so the `args_count` must not exceed 2.

After checking the content of `sys.argv`, you create a `pathlib.Path` object to store the path to your target directory. If this directory doesn't exist, then you inform the user and exit the app. The `for` loop lists the directory content, one entry per line.

If you [run the script](#) from your command line, then you'll get the following results:

## Shell

```
$ python ls_argv.py sample/  
hello.txt  
lorem.md  
realpython.md  
  
$ python ls_argv.py  
You must specify the target directory  
  
$ python ls_argv.py sample/ other_dir/  
One argument expected, got 2  
  
$ python ls_argv.py non_existing/  
The target directory doesn't exist
```

Your program takes a directory as an argument and lists its content. If you run the command without arguments, then you get an error message. If you run the command with more than one target directory, you also get an error. Running the command with a nonexistent directory produces another error message.

Even though your program works okay, parsing command-line arguments manually using the `sys.argv` attribute isn't a scalable solution for more complex CLI apps. If your app needs to take many more arguments and options, then parsing `sys.argv` will be a complex and error-prone task. You need something better, and you get it in Python's `argparse` module.

## Creating a CLI With `argparse`

A much more convenient way to create CLI apps in Python is using the [argparse](#) module, which comes in the [standard library](#). This module was first released in [Python 3.2](#) with [PEP 389](#) and is a quick way to create CLI apps in Python without installing a third-party library, such as [Typer](#) or [Click](#).

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

Python's `argparse` module allows you to:

- Parse command-line **arguments** and **options**
- Take a **variable number of parameters** in a single option
- Provide **subcommands** in your CLIs

These features turn `argparse` into a powerful CLI framework that you can confidently rely on when creating your CLI applications. To use Python's `argparse`, you'll need to follow four straightforward steps:

1. Import `argparse`.
2. Create an **argument parser** by instantiating `ArgumentParser`.
3. Add **arguments** and **options** to the parser using the `.add_argument()` method.
4. Call `.parse_args()` on the parser to get the `Namespace` of arguments.

As an example, you can use `argparse` to improve your `ls_argv.py` script. Go ahead and create `ls.py` with the following code:

## Python

```
# ls.py v1

import argparse
from pathlib import Path

parser = argparse.ArgumentParser()

parser.add_argument("path")

args = parser.parse_args()

target_dir = Path(args.path)

if not target_dir.exists():
    print("The target directory doesn't exist")
    raise SystemExit(1)

for entry in target_dir.iterdir():
    print(entry.name)
```

Your code has changed significantly with the introduction of `argparse`. The most notable difference from the previous version is that the [conditional statements](#) to check the arguments provided by the user are gone. That's because `argparse` automatically checks the presence of arguments for you.

In this new implementation, you first import `argparse` and create an argument parser. To create the parser, you use the `ArgumentParser` class. Next, you define an argument called `path` to get the user's target directory.

The next step is to call `.parse_args()` to parse the input arguments and get a `Namespace` object that contains all the user's arguments. Note that now the `args` [variable](#) holds a

Namespace object, which has a property for each argument that's been gathered from the command line.

In this example, you only have one argument, called `path`. The Namespace object allows you to access `path` using the **dot notation** on `args`. The rest of your code remains the same as in the first implementation.

Now go ahead and run this new script from your command line:

#### Shell

```
$ python ls.py sample/
lorem.md
realpython.md
hello.txt

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

$ python ls.py sample/ other_dir/
usage: ls.py [-h] path
ls.py: error: unrecognized arguments: other_dir/

$ python ls.py non_existing/
The target directory doesn't exist
```

The first command [prints](#) the same output as your original script, `ls_argv.py`. In contrast, the second command displays output that's quite different from in `ls_argv.py`. The program now shows a usage message and issues an error telling you that you must provide the `path` argument.

In the third command, you pass two target directories, but the app isn't prepared for that. Therefore, it shows the usage message again and throws an error letting you know about the underlying problem.

Finally, if you run the script with a nonexistent directory as an argument, then you get an error telling you that the target directory doesn't exist, so the program can't do its work.

A new implicit feature is now available to you. Now your program accepts an optional `-h` flag. Go ahead and give it a try:

#### Shell

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

positional arguments:
  path

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

Great, now your program automatically responds to the `-h` or `--help` flag, displaying a help message with usage instructions for you. That's a really neat feature, and you get it for free by introducing `argparse` into your code!

With this quick introduction to creating CLI apps in Python, you're now ready to dive deeper into the `argparse` module and all its cool features.

### 5 Thoughts on Mastering Python

A free email class for Python developers  
[realpython.com](https://realpython.com)



 Remove ads

## Creating Command-Line Interfaces With Python's `argparse`

You can use the `argparse` module to write user-friendly command-line interfaces for your applications and projects. This module allows you to define the arguments and options that

applications and projects. This module allows you to define the arguments and options that your app will require. Then `argparse` will take care of parsing those arguments and options of `sys.argv` for you.

Another cool feature of `argparse` is that it automatically generates usage and help messages for your CLI apps. The module also issues errors in response to invalid arguments and more.

Before diving deeper into `argparse`, you need to know that the module's [documentation](#) recognizes two different types of command-line arguments:

1. **Positional arguments**, which you know as arguments
2. **Optional arguments**, which you know as options, flags, or switches

In the `ls.py` example, `path` is a **positional argument**. Such an argument is called *positional* because its relative position in the command construct defines its purpose.

**Optional** arguments aren't mandatory. They allow you to modify the behavior of the command. In the `ls` Unix command example, the `-l` flag is an optional argument, which makes the command display a detailed output.

With these concepts clear, you can kick things off and start building your own CLI apps with Python and `argparse`.

## Creating a Command-Line Argument Parser

The command-line argument parser is the most important part of any `argparse` CLI. All the arguments and options that you provide at the command line will pass through this parser, which will do the hard work for you.

To create a command-line argument parser with `argparse`, you need to instantiate the [ArgumentParser](#) class:

Python

>>>

```
>>> from argparse import ArgumentParser

>>> parser = ArgumentParser()
>>> parser
ArgumentParser(
  prog='',
  usage=None,
  description=None,
  formatter_class=<class 'argparse.HelpFormatter'>,
  conflict_handler='error',
  add_help=True
)
```

The [constructor](#) of `ArgumentParser` takes many different arguments that you can use to tweak several features of your CLIs. All of its arguments are optional, so the most bare-bones parser that you can create results from instantiating `ArgumentParser` without any arguments.

You'll learn more about the arguments to the `ArgumentParser` constructor throughout this tutorial, particularly in the section on [customizing your argument parser](#). For now, you can tackle the next step in creating a CLI with `argparse`. That step is to add arguments and options through the parser object.

## Adding Arguments and Options

To add arguments and options to an `argparse` CLI, you'll use the `.add_argument()` method of your `ArgumentParser` instance. Note that the method is common for arguments and options. Remember that in the `argparse` terminology, arguments are called **positional arguments**, and options are known as **optional arguments**.



The first argument to the `.add_argument()` method sets the difference between arguments and options. This argument is identified as either `name` or `flag`. So, if you provide a name, then you'll be defining an argument. In contrast, if you use a flag, then you'll add an option.

You've already worked with command-line arguments in `argparse`. So, consider the following enhanced version of your custom `ls` command, which adds an `-l` option to the CLI:

## Python

```
1 # ls.py v2
2
3 import argparse
4 import datetime
5 from pathlib import Path
6
7 parser = argparse.ArgumentParser()
8
9 parser.add_argument("path")
10
11 parser.add_argument("-l", "--long", action="store_true")
12
13 args = parser.parse_args()
14
15 target_dir = Path(args.path)
16
17 if not target_dir.exists():
18     print("The target directory doesn't exist")
19     raise SystemExit(1)
20
21 def build_output(entry, long=False):
22     if long:
23         size = entry.stat().st_size
24         date = datetime.datetime.fromtimestamp(
25             entry.stat().st_mtime).strftime(
26                 "%b %d %H:%M:%S")
27     )
28     return f"{size:>6d} {date} {entry.name}"
29     return entry.name
30
31 for entry in target_dir.iterdir():
32     print(build_output(entry, long=args.long))
```

In this example, line 11 creates an option with the flags `-l` and `--long`. The syntactical

In this example, line 11 creates an option with the flags `-l` and `--long`. The syntactical difference between arguments and options is that option names start with `-` for shorthand flags and `--` for long flags.

Note that in this specific example, an action argument set to `"store_true"` accompanies the `-l` or `--long` option, which means that this option will store a [Boolean](#) value. If you provide the option at the command line, then its value will be `True`. If you miss the option, then its value will be `False`. You'll learn more about the action argument to `.add_argument()` in the [Setting the Action Behind an Option](#) section.

The `build_output()` function on line 21 [returns](#) a detailed output when `long` is `True` and a minimal output otherwise. The detailed output will contain the size, modification date, and name of all the entries in the target directory. It uses tools like the `Path.stat()` and a `datetime.datetime` object with a custom [string](#) format.

Go ahead and execute your program on `sample` to check how the `-l` option works:

#### Shell

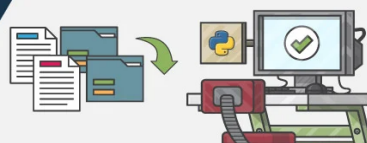
```
$ python ls.py -l sample/
2609 Oct 28 14:07:04 lorem.md
428 Oct 28 14:07:04 realpython.md
83 Oct 28 14:07:04 hello.txt
```

Your new `-l` option allows you to generate and display a more detailed output about the content of your target directory.

Now that you know how to add command-line arguments and options to your CLIs, it's time to dive into parsing those arguments and options. That's what you'll explore in the following section.

**Free PDF Download: Python 3 Cheat Sheet**

[Download Now](#)  
realpython.com



## Parsing Command-Line Arguments and Options

Parsing the command-line arguments is another important step in any CLI app based on `argparse`. Once you've parsed the arguments, then you can start taking action in response to their values. In your custom `ls` command example, the argument parsing happens on the line containing the `args = parser.parse_args()` statement.

This statement calls the `.parse_args()` method and assigns its return value to the `args` variable. The return value of `.parse_args()` is a `Namespace` object containing all the arguments and options provided at the command line and their corresponding values.

Consider the following toy example:

Python

>>>

```
>>> from argparse import ArgumentParser

>>> parser = ArgumentParser()

>>> parser.add_argument("site")
_StoreAction(...)

>>> parser.add_argument("-c", "--connect", action="store_true")
_StoreTrueAction(...)

>>> args = parser.parse_args(["Real Python", "-c"])
>>> args
Namespace(site='Real Python', connect=True)

>>> args.site
'Real Python'
>>> args.connect
True
```

The `Namespace` object that results from calling `.parse_args()` on the command-line argument parser gives you access to all the input arguments, options, and their

argument parser gives you access to all the input arguments, options, and their corresponding values by using the **dot notation**. This way, you can check the list of input

arguments and options to take actions in response to the user's choices at the command line.

You'll use this Namespace object in your application's main code. That's what you did under the `for` loop in your custom `ls` command example.

Up to this point, you've learned about the main steps for creating argparse CLIs. Now you can take some time to learn the basics of how to organize and build a CLI application in Python.

## Setting Up Your CLI App's Layout and Build System

Before continuing with your argparse learning adventure, you should pause and think of how you would organize your code and [lay out](#) a CLI project. First, you should observe the following points:

- You can create [modules and packages](#) to organize your code.
- You can name the core package of a Python app after the app itself.
- You'll name each Python module according to its specific content or functionality.
- You can add a `__main__.py` module to any Python package if you want to make that package directly executable.

With these ideas in mind and considering that the [model-view-controller \(MVC\)](#) pattern is an effective way to structure your applications, you can use the following directory structure when laying out a CLI project:

```
hello_cli/
|
├─ hello_cli/
|   ├── __init__.py
|   ├── __main__.py
|   ├── cli.py
|   └── model.py
|
├─ tests/
|   ├── __init__.py
|   ├── test_cli.py
|   └── test_model.py
|
├─ pyproject.toml
├─ README.md
└─ requirements.txt
```

The `hello_cli/` directory is the project's root directory. There, you'll place the following files:

- `pyproject.toml` is a [TOML](#) file that specifies the project's **build system** and other **configurations**.
- `README.md` provides the project **description** and **instructions** for installing and running the application. Adding a descriptive and detailed `README.md` file to your projects is a best practice in programming, especially if you're planning to release the project as an open-source solution.
- `requirements.txt` provides a conventional file that lists the project's **external dependencies**. You'll use this file to automatically install the dependencies using `pip` with the `-r` option.

Then you have the `hello_cli/` directory that holds the app's core package, which contains

the following modules:

- `__init__.py` enables `hello_cli/` as a Python **package**.
- `__main__.py` provides the application's **entry-point script** or executable file.
- `cli.py` provides the application's command-line interface. The code in this file will play the **view-controller** role in the MVC-based architecture.
- `model.py` contains the code that supports the app's main functionalities. This code will play the **model** role in your MVC layout.

You'll also have a `tests/` package containing files with [unit tests](#) for your app's components. In this specific project layout example, you have `test_cli.py` for unit tests that check the CLI's functionality and `test_model.py` for unit tests that check your model's code.

The `pyproject.toml` file allows you to define the app's build system as well as many other general configurations. Here's a minimal example of how to fill in this file for your sample `hello_cli` project:

TOML

```
# pyproject.toml

[build-system]
requires = ["setuptools>=64.0.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "hello_cli"
version = "0.0.1"
description = "My awesome Hello CLI application"
readme = "README.md"
authors = [{ name = "Real Python", email = "info@realpython.com" }]

[project.scripts]
hello_cli = "hello_cli.__main__:main"
```

The `[build-system]` table header sets up `setuptools` as your app's [build system](#) and

The `[build-system]` [table header](#) sets up `setuptools` as your app's `build system` and specifies which dependencies Python needs to install for building your app. The `[project]` header provides general metadata for your application. This metadata is pretty useful when you want to [publish your app](#) to the Python package index ([PyPI](#)). Finally, the `[project.scripts]` heading defines the entry point to your application.

With this quick dive into laying out and building CLI projects, you're ready to continue learning about `argparse`, especially how to customize your command-line argument parser.

## Python Dependency Management Pitfalls

A free email class  
[realpython.com](https://realpython.com)



 Remove ads

# Customizing Your Command-Line Argument Parser

In previous sections, you learned the basics of using Python's `argparse` to implement command-line interfaces for your programs or applications. You also learned how to organize and lay out a CLI app project following the MVC pattern.

In the following sections, you'll dive deeper into many other neat features of `argparse`. Specifically, you'll learn how to use some of the most useful arguments in the `ArgumentParser` constructor, which will allow you to customize the general behavior of your CLI apps.

## Tweaking the Program's Help and Usage Content

Providing usage instructions and help to the users of your CLI applications is a best practice that'll make your users' lives more pleasant with a great [user experience \(UX\)](#). In this section, you'll learn how to take advantage of some arguments of `ArgumentParser` to fine-tune how your CLI apps show help and usage messages to their users. You'll learn how to



tune how your CLI apps show help and usage messages to their users. You'll learn how to:

- Set the program's name
- Define the program's description and epilog message
- Display grouped help for arguments and options

To kick things off, you'll start by setting your program's name and specifying how that name will look in the context of a help or usage message.

## Setting the Program's Name

By default, `argparse` uses the first value in `sys.argv` to set the program's name. This first item holds the name of the Python file that you've just executed. This filename will look odd in a usage message.

As an example, go ahead and run your custom `ls` command with the `-h` option:

### Shell

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

positional arguments:
  path

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

The highlighted line in the command's output shows that `argparse` is using the filename `ls.py` as the program's name. This looks odd because app names rarely include file extensions when displayed in usage messages.

Fortunately, you can specify the name of your program by using the `prog` argument like in the following code snippet:

## Python

```
# ls.py v3

import argparse
import datetime
from pathlib import Path

parser = argparse.ArgumentParser(prog="ls")

# ...

for entry in target_dir.iterdir():
    print(build_output(entry, long=args.long))
```

With the prog argument, you specify the program name that'll be used in the usage message. In this example, you use the "ls" string. Now go ahead and run your app again:

## Shell

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

positional arguments:
  path

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

Great! The app's usage message in the first line of this output shows ls instead of ls.py as the program's name.

Apart from setting the program's name, argparse lets you define the app's description and

epilog message. You'll learn how to do both in the following section.

## Define the Program's Description and Epilog Message

You can also define a general description for your application and an epilog or closing message. To do this, you'll use the `description` and `epilog` arguments, respectively. Go ahead and update the `ls.py` file with the following additions to the `ArgumentParser` constructor:

Python

```
# ls.py v4

import argparse
import datetime
from pathlib import Path

parser = argparse.ArgumentParser(
    prog="ls",
    description="List the content of a directory",
    epilog="Thanks for using %(prog)s! :)",
)

# ...

for entry in target_dir.iterdir():
    print(build_output(entry, long=args.long))
```

In this update, `description` allows you to provide a general description for your app. This description will display at the beginning of the help message. The `epilog` argument lets you define some text as your app's epilog or closing message. Note that you can interpolate the `prog` argument into the epilog string using the [old-style string-formatting operator \(%\)](#).

**Note:** Help messages support **format specifiers** of the form `%(specifier)s`. These specifiers use the string formatting operator, `%`, rather than the popular [f-strings](#).

That's because f-strings replace names with their values immediately as they run.

Therefore, inserting `prog` into `epilog` in the call to `ArgumentParser` above will fail with a `NameError` if you use an f-string.

If you run the app again, then you'll get an output like the following:

#### Shell

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

```
List the content of a directory
```

```
positional arguments:
  path
```

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

```
Thanks for using ls! :)
```

Now the output shows the description message right after the usage message and the epilog message at the end of the help text.

## Display Grouped Help for Arguments and Options

**Help groups** are another interesting feature of `argparse`. They allow you to group related commands and arguments, which will help you organize the app's help message. To create these help groups, you'll use the `.add_argument_group()` method of `ArgumentParser`.

As an example, consider the following updated version of your custom `ls` command:

## Python

```
# ls.py v5
# ...

parser = argparse.ArgumentParser(
    prog="ls",
    description="List the content of a directory",
    epilog="Thanks for using %(prog)s! :)",
)

general = parser.add_argument_group("general output")
general.add_argument("path")

detailed = parser.add_argument_group("detailed output")
detailed.add_argument("-l", "--long", action="store_true")

args = parser.parse_args()

# ...

for entry in target_dir.iterdir():
    print(build_output(entry, long=args.long))
```

In this update, you create a help group for arguments and options that display general output and another group for arguments and options that display detailed output.

**Note:** In this specific example, grouping arguments like this may seem unnecessary. However, if your app has several arguments and options, then using help groups can significantly improve your user experience.

If you run the app with the `-h` option at your command line, then you'll get the following output:

## Shell

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

List the content of a directory

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

general output:
  path

detailed output:
  -l, --long

Thanks for using ls! :)
```

Now your app's arguments and options are conveniently grouped under descriptive headings in the help message. This neat feature will help you provide more context to your users and improve their understanding of how the app works.



 [The Real Python Podcast »](#)

 [Remove ads](#)

## Providing Global Settings for Arguments and Options

Beyond customizing the usage and help messages, `ArgumentParser` also allows you to perform a few other interesting tweaks to your CLI apps. Some of these tweaks include:

- Defining a global default value for arguments and options
- Loading arguments and options from an external file
- Allowing or disallowing option abbreviations

Sometimes, you may need to specify a single **global default value** for your app's arguments and options. You can do this by passing the default value to `argument_default` on the call to the `ArgumentParser` constructor.

This feature may be only rarely useful because arguments and options often have a different data type or meaning, and it can be difficult to find a value that suits all the requirements.

However, a common use case of `argument_default` is when you want to avoid adding arguments and options to the `Namespace` object. In this situation, you can use the `SUPPRESS` constant as the default value. This default value will make it so that only the arguments and options provided at the command line end up stored in the arguments `Namespace`.

As an example, go ahead and modify your custom `ls` command as in the snippet below:

## Python

```
# ls.py v6

import argparse
import datetime
from pathlib import Path

parser = argparse.ArgumentParser(
    prog="ls",
    description="List the content of a directory",
    epilog="Thanks for using %(prog)s! :)",
    argument_default=argparse.SUPPRESS,
)

# ...

for entry in target_dir.iterdir():
    try:
        long = args.long
    except AttributeError:
        long = False
    print(build_output(entry, long=long))
```

By passing `SUPPRESS` to the `ArgumentParser` constructor, you prevent non-supplied arguments from being stored in the arguments `Namespace` object. That's why you have to check if the `-l` or `--long` option was actually passed before calling `build_output()`. Otherwise, your code will break with an `AttributeError` because `long` won't be present in `args`.

Another cool feature of `ArgumentParser` is that it allows you to *load argument values from an external file*. This possibility comes in handy when you have an application with long or complicated command-line constructs, and you want to automate the process of loading argument values.



argument values.

In this situation, you can store the argument values in an external file and ask your program to load them from it. To try this feature out, go ahead and create the following toy CLI app:

Python

```
# fromfile.py

import argparse

parser = argparse.ArgumentParser(fromfile_prefix_chars="@")

parser.add_argument("one")
parser.add_argument("two")
parser.add_argument("three")

args = parser.parse_args()

print(args)
```

Here, you pass the @ symbol to the `fromfile_prefix_chars` argument of `ArgumentParser`. Then you create three required arguments that must be provided at the command line.

Now say that you often use this application with the same set of argument values. To facilitate and streamline your work, you can create a file containing appropriate values for all the necessary arguments, one per line, like in the following `args.txt` file:

Text

```
first
second
third
```

With this file in place, you can now call your program and instruct it to load the values from

the `args.txt` file like in the following command run:

#### Shell

```
$ python fromfile.py @args.txt
Namespace(one='first', two='second', three='third')
```

In this command's output, you can see that `argparse` has read the content of `args.txt` and sequentially assigned values to each argument of your `fromfile.py` program. All the arguments and their values are successfully stored in the `Namespace` object.

The ability to accept **abbreviated option names** is another cool feature of `argparse` CLIs. This feature is enabled by default and comes in handy when your program has long option names. As an example, consider the following program, which prints out the value that you specify at the command line under the `--argument-with-a-long-name` option:

#### Python

```
# abbreviate.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--argument-with-a-long-name")

args = parser.parse_args()

print(args.argument_with_a_long_name)
```

This program prints whatever you pass as an argument to the `--argument-with-a-long-name` option. Go ahead and run the following commands to check how the Python `argparse` module handles abbreviations for you:

## Shell

```
$ python abbreviate.py --argument-with-a-long-name 42
42

$ python abbreviate.py --argument 42
42

$ python abbreviate.py --a 42
42
```

These examples show how you can abbreviate the name of the `--argument-with-a-long-name` option and still get the app to work correctly. This feature is enabled by default. If you want to disable it and forbid abbreviations, then you can use the `allow_abbrev` argument to `ArgumentParser`:

## Python

```
# abbreviate.py

import argparse

parser = argparse.ArgumentParser(allow_abbrev=False)

parser.add_argument("--argument-with-a-long-name")

args = parser.parse_args()

print(args.argument_with_a_long_name)
```

Setting `allow_abbrev` to `False` disables abbreviations in command-line options. From this point on, you'll have to provide the complete option name for the program to work correctly. Otherwise, you'll get an error:

### Shell

```
$ python abbreviate.py --argument-with-a-long-name 42
42

$ python abbreviate.py --argument 42
usage: abbreviate.py [-h] [--argument-with-a-long-name ...]
abbreviate.py: error: unrecognized arguments: --argument 42
```

The error message in the second example tells you that the `--argument` option isn't recognized as a valid option. To use the option, you need to provide its full name.

### Find Your Dream Python Job

[pythonjobshq.com](https://pythonjobshq.com)



 Remove ads

## Fine-Tuning Your Command-Line Arguments and Options

Up to this point, you've learned how to customize several features of the `ArgumentParser` class to improve the user experience of your CLIs. Now you know how to tweak the usage and help messages of your apps and how to fine-tune some global aspects of command-line arguments and options.

In this section, you'll learn how to customize several other features of your CLI's command-line arguments and options. In this case, you'll be using the `.add_argument()` method and some of its most relevant arguments, including `action`, `type`, `nargs`, `default`, `help`, and a few others.

# Setting the Action Behind an Option

When you add an option or flag to a command-line interface, you'll often need to define how you want to store the option's value in the `Namespace` object that results from calling `.parse_args()`. To do this, you'll use the `action` argument to `.add_argument()`. The `action` argument defaults to `"store"`, which means that the value provided for the option at hand will be stored as is in the `Namespace`.

The `action` argument can take one of several possible values. Here's the list of these possible values and their meanings:

Allowed Value	Description
<code>store</code>	Stores the input value to the <code>Namespace</code> object
<code>store_const</code>	Stores a constant value when the option is specified
<code>store_true</code>	Stores the <code>True</code> <a href="#">Boolean value</a> when the option is specified and stores <code>False</code> otherwise
<code>store_false</code>	Stores <code>False</code> when the option is specified and stores <code>True</code> otherwise
<code>append</code>	<a href="#">Appends</a> the current value to a <a href="#">list</a> each time the option is provided
<code>append_const</code>	Appends a constant value to a list each time the option is provided
<code>count</code>	Stores the number of times the current option has been provided
<code>version</code>	Shows the app's version and terminates the execution

In this table, the values that include the `_const` suffix in their names require you to provide the desired constant value using the `const` argument in the call to the `.add_argument()`

method. Similarly, the `version` action requires you to provide the app's version by passing the `version` argument to `.add_argument()`. You should also note that only the `store` and `append` actions can and must take arguments at the command line.

To try these actions out, you can create a toy app with the following implementation:

#### Python

```
# actions.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument(
    "--name", action="store"
) # Equivalent to parser.add_argument("--name")
parser.add_argument("--pi", action="store_const", const=3.14)
parser.add_argument("--is-valid", action="store_true")
parser.add_argument("--is-invalid", action="store_false")
parser.add_argument("--item", action="append")
parser.add_argument("--repeated", action="append_const", const=42)
parser.add_argument("--add-one", action="count")
parser.add_argument(
    "--version", action="version", version="% (prog)s 0.1.0"
)

args = parser.parse_args()

print(args)
```

This program implements an option for each type of action discussed above. Then the program prints the resulting Namespace of arguments. Here's a summary of how these options will work:

- `--name` will store the value passed, without any further consideration.
- `--pi` will automatically store the target constant when the option is provided.
- `--is-valid` will store `True` when provided and `False` otherwise. If you need the opposite behavior, use a `store_false` action like `--is-invalid` in this example.
- `--item` will let you create a list of all the values. You must repeat the option for each value. Under the hood, `argparse` will append the items to a list named after the option itself.
- `--repeated` will work similarly to `--item`. However, it always appends the same constant value, which you must provide using the `const` argument.
- `--add-one` counts how many times the option is passed at the command line. This type of option is quite useful when you want to implement several verbosity levels in your programs. For example, `-v` can mean level one of verbosity, `-vv` may indicate level two, and so on.
- `--version` shows the app's version and terminates the execution immediately. Note that you must provide the version number beforehand, which you can do by using the `version` argument when creating the option with `.add_argument()`.

Go ahead and run the script with the following command construct to try out all these options:



Linux + macOS

### Windows PowerShell

```
PS> python actions.py `
> --name Python `
> --pi `
> --is-valid `
> --is-invalid `
> --item 1 --item 2 --item 3 `
> --repeat --repeat --repeat `
> --add-one --add-one --add-one
Namespace(
  name='Python',
  pi=3.14,
  is_valid=True,
  is_invalid=False,
  item=['1', '2', '3'],
  repeated=[42, 42, 42],
  add_one=3
)

PS> python actions.py --version
actions.py 0.1.0
```

With this command, you show how all the actions work and how they're stored in the resulting Namespace object. The version action is the last one that you used, because this option just shows the version of the program and then ends the execution. It doesn't get stored in the Namespace object.

Even though the default set of actions is quite complete, you also have the possibility of creating custom actions by subclassing the `argparse.Action` class. If you decide to do this, then you must override the `.__call__()` method, which turns instances into callable objects. Optionally, you can override the `.__init__()` and `.format_usage()` methods depending on your needs.



To override the `.__call__()` method, you need to ensure that the method's signature includes the `parser`, `namespace`, `values`, and `option_string` arguments.

In the following example, you implement a minimal and verbose store action that you can use when building your CLI apps:

#### Python

```
# custom_action.py

import argparse

class VerboseStore(argparse.Action):
    def __call__(self, parser, namespace, values, option_string=None):
        print(f"Storing {values} in the {option_string} option...")
        setattr(namespace, self.dest, values)

parser = argparse.ArgumentParser()

parser.add_argument("-n", "--name", action=VerboseStore)

args = parser.parse_args()

print(args)
```

In this example, you define `VerboseStore` inheriting from `argparse.Action`. Then you override the `.__call__()` method to print an informative message and set the target option in the namespace of command-line arguments. Finally, the app prints the namespace itself.

Go ahead and run the following command to try out your custom action:

#### Shell

```
$ python custom_action.py --name Python
Storing Python in the --name option...
Namespace(name='Python')
```

Great! Your program now prints out a message before storing the value provided to the `--name` option at the command line. Custom actions like the one in the above example allow you to fine-tune how your programs' options are stored.

To continue fine-tuning your argparse CLIs, you'll learn how to customize the input value of command-line arguments and options in the following section.



[Learn Python »](#)

 Remove ads

## Customizing Input Values in Arguments and Options

Another common requirement when you're building CLI applications is to customize the input values that arguments and options will accept at the command line. For example, you may require that a given argument accept an integer value, a list of values, a string, and so on.

By default, any argument provided at the command line will be treated as a string. Fortunately, argparse has internal mechanisms to check if a given argument is a valid integer, string, list, and more.

In this section, you'll learn how to customize the way in which argparse processes and stores input values. Specifically, you'll learn how to:

- Set the **data type** of input values for arguments and options
- Take **multiple input values** in arguments and options
- Provide **default values** for arguments and options
- Define a list of **allowed input values** for arguments and options

To kick things off, you'll start by customizing the data type that your arguments and

To kick things off, you'll start by customizing the data type that your arguments and options will accept at the command line.

## Setting the Type of Input Values

When creating argparse CLIs, you can define the type that you want to use when storing command-line arguments and options in the Namespace [object](#). To do this, you can use the type argument of `.add_argument()`.

As an example, say that you want to write a sample CLI app for dividing two [numbers](#). The app will take two options, `--dividend` and `--divisor`. These options will only accept integer numbers at the command line:

Python

```
# divide.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--dividend", type=int)
parser.add_argument("--divisor", type=int)

args = parser.parse_args()

print(args.dividend / args.divisor)
```

In this example, you set the type of `--dividend` and `--divisor` to `int`. This setting will make your options only accept valid integer values as input. If the input value can't be converted to the `int` type without losing information, then you'll get an error:

## Shell

```
$ python divide.py --dividend 42 --divisor 2
21.0

$ python divide.py --dividend "42" --divisor "2"
21.0

$ python divide.py --dividend 42 --divisor 2.0
usage: divide.py [-h] [--dividend DIVIDEND] [--divisor DIVISOR]
divide.py: error: argument --divisor: invalid int value: '2.0'

$ python divide.py --dividend 42 --divisor two
usage: divide.py [-h] [--dividend DIVIDEND] [--divisor DIVISOR]
divide.py: error: argument --divisor: invalid int value: 'two'
```

The first two examples work correctly because the input values are integer numbers. The third example fails with an error because the divisor is a floating-point number. The last example also fails because two isn't a numeric value.

## Taking Multiple Input Values

Taking multiple values in arguments and options may be a requirement in some of your CLI applications. By default, `argparse` assumes that you'll expect a single value for each argument or option. You can modify this behavior with the `nargs` argument of `.add_argument()`.

The `nargs` argument tells `argparse` that the underlying argument can take zero or more input values depending on the specific value assigned to `nargs`. If you want the argument or option to accept a fixed number of input values, then you can set `nargs` to an integer number. If you need more flexible behaviors, then `nargs` has you covered because it also accepts the following values:

Allowed Value	Meaning
?	Accepts a single input value, which can be optional
*	Takes zero or more input values, which will be stored in a list
+	Takes one or more input values, which will be stored in a list
<code>argparse.REMAINDER</code>	Gathers all the values that are remaining in the command line

It's important to note that this list of allowed values for nargs works for both command-line arguments and options.

To start trying out the allowed values for nargs, go ahead and create a `point.py` file with the following code:

#### Python

```
# point.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--coordinates", nargs=2)

args = parser.parse_args()

print(args)
```

In this small app, you create a command-line option called `--coordinates` that takes two input values representing the x and y [Cartesian](#) coordinates. With this script in place, go ahead and run the following commands:

## Shell

```
$ python point.py --coordinates 2 3
Namespace(coordinates=['2', '3'])

$ python point.py --coordinates 2
usage: point.py [-h] [--coordinates COORDINATES COORDINATES]
point.py: error: argument --coordinates: expected 2 arguments

$ python point.py --coordinates 2 3 4
usage: point.py [-h] [--coordinates COORDINATES COORDINATES]
point.py: error: unrecognized arguments: 4

$ python point.py --coordinates
usage: point.py [-h] [--coordinates COORDINATES COORDINATES]
point.py: error: argument --coordinates: expected 2 arguments
```

In the first command, you pass two numbers as input values to `--coordinates`. In this case, the program works correctly, storing the values in a list under the `coordinates` attribute in the `Namespace` object.

In the second example, you pass a single input value, and the program fails. The error message tells you that the app was expecting two arguments, but you only provided one. The third example is pretty similar, but in that case, you supplied more input values than required.

The final example also fails because you didn't provide input values at all, and the `--coordinates` option requires two values. In this example, the two input values are mandatory.

To try out the `*` value of `nargs`, say that you need a CLI app that takes a list of numbers at the command line and returns their sum:

## Python

```
# sum.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("numbers", nargs="*", type=float)

args = parser.parse_args()

print(sum(args.numbers))
```

The numbers argument accepts zero or more floating-point numbers at the command line because you've set nargs to \*. Here's how this script works:

## Shell

```
$ python sum.py 1 2 3
6.0

$ python sum.py 1 2 3 4 5 6
21.0

$ python sum.py
0
```

The first two commands show that numbers accepts an undetermined number of values at the command line. These values will be stored in a list named after the argument itself in the Namespace object. If you don't pass any values to sum.py, then the corresponding list of values will be empty, and the sum will be 0.

Next up, you can try the + value of nargs with another small example. This time, say that

you need an app that accepts one or more files at the command line. You can code this app like in the example below:

#### Python

```
# files.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("files", nargs="+")

args = parser.parse_args()

print(args)
```

The files argument in this example will accept one or more values at the command line. You can give it a try by running the following commands:

#### Shell

```
$ python files.py hello.txt
Namespace(files=['hello.txt'])

$ python files.py hello.txt realpython.md README.md
Namespace(files=['hello.txt', 'realpython.md', 'README.md'])

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

The first two examples show that files accepts an undefined number of files at the command line. The last example shows that you can't use files without providing a file, as you'll get an error. This behavior forces you to provide at least one file to the files argument.



The final allowed value for nargs is REMAINDER. This constant allows you to capture the remaining values provided at the command line. If you pass this value to nargs, then the

underlying argument will work as a bag that'll gather all the extra input values. As an exercise, go ahead and explore how REMAINDER works by coding a small app by yourself.

Even though the nargs argument gives you a lot of flexibility, sometimes it's pretty challenging to use this argument correctly in multiple command-line options and arguments. For example, it can be hard to reliably combine arguments and options with nargs set to \*, +, or REMAINDER in the same CLI:

#### Python

```
# cooking.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("veggies", nargs="+")
parser.add_argument("fruits", nargs="*")

args = parser.parse_args()

print(args)
```

In this example, the veggies argument will accept one or more vegetables, while the fruits argument should accept zero or more fruits at the command line. Unfortunately, this example doesn't work as expected:

#### Shell

```
$ python cooking.py pepper tomato apple banana
Namespace(veggies=['pepper', 'tomato', 'apple', 'banana'], fruits=[])
```

The command's output shows that all the provided input values have been stored in the

veggies attribute, while the fruits attribute holds an empty list. This happens because the argparse parser doesn't have a reliable way to determine which value goes to which

argument or option. In this specific example, you can fix the problem by turning both arguments into options:

#### Python

```
# cooking.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--veggies", nargs="+")
parser.add_argument("--fruits", nargs="*")

args = parser.parse_args()

print(args)
```

With this minor update, you're ensuring that the parser will have a secure way to parse the values provided at the command line. Go ahead and run the following command to confirm this:

#### Shell

```
$ python cooking.py --veggies pepper tomato --fruits apple banana
Namespace(veggies=['pepper', 'tomato'], fruits=['apple', 'banana'])
```

Now each input value has been stored in the correct list in the resulting Namespace. The argparse parser has used the option names to correctly parse each supplied value.

To avoid issues similar to the one discussed in the above example, you should always be careful when trying to combine arguments and options with nargs set to \*, +, or REMAINDER.

## Providing Default Values

The `.add_argument()` method can take a `default` argument that allows you to provide an appropriate default value for individual arguments and options. This feature can be useful when you need the target argument or option to always have a valid value in case the user doesn't provide any input at the command line.

As an example, get back to your custom `ls` command and say that you need to make the command list the content of the current directory when the user doesn't provide a target directory. You can do this by setting `default` to `"."` like in the code below:

Python

```
# ls.py v7

import argparse
import datetime
from pathlib import Path

# ...

general = parser.add_argument_group("general output")
general.add_argument("path", nargs="?", default=".")

# ...
```

The highlighted line in this code snippet does the magic. In the call to `.add_argument()`, you use `nargs` with the question mark (?) as its value. You need to do this because all the command-line arguments in `argparse` are required, and setting `nargs` to either `?`, `*`, or `+` is the only way to skip the required input value. In this specific example, you use `?` because you need a single input value or none.

Then you set `default` to the `"."` string, which represents the current working directory. With these updates, you can now run `ls.py` without providing a target directory. It'll list the

content of its default directory. To try it out, go ahead and run the following commands:

#### Shell

```
$ cd sample/  
  
$ python ../ls.py  
lorem.md  
realpython.md  
hello.txt
```

Now your custom `ls` command lists the current directory's content if you don't provide a target directory at the command line. Isn't that cool?

## Specifying a List of Allowed Input Values

Another interesting possibility in `argparse` CLIs is that you can create a domain of allowed values for a specific argument or option. You can do this by providing a list of accepted values using the `choices` argument of `.add_argument()`.

Here's an example of a small app with a `--size` option that only accepts a few predefined input values:

#### Python

```
# size.py  
  
import argparse  
  
parser = argparse.ArgumentParser()  
  
parser.add_argument("--size", choices=["S", "M", "L", "XL"], default="M")  
  
args = parser.parse_args()  
  
print(args)
```

In this example, you use the `choices` argument to provide a list of allowed values for the `--size` option. This setting will cause the option to only accept the predefined values. If you try to use a value that's not in the list, then you get an error:

#### Shell

```
$ python size.py --size S
Namespace(size='S')

$ python choices.py --size A
usage: choices.py [-h] [--size {S,M,L,XL}]
choices.py: error: argument --size: invalid choice: 'A'
        (choose from 'S', 'M', 'L', 'XL')
```

If you use an input value from the list of allowed values, then your app works correctly. If you use an extraneous value, then the app fails with an error.

The `choices` argument can hold a list of allowed values, which can be of different data types. For integer values, a useful technique is to use a range of accepted values. To do this, you can use `range()` like in the following example:

#### Python

```
# weekdays.py

import argparse

my_parser = argparse.ArgumentParser()

my_parser.add_argument("--weekday", type=int, choices=range(1, 8))

args = my_parser.parse_args()

print(args)
```

In this example, the value provided at the command line will be automatically checked against the range object provided as the choices argument. Go ahead and give this example a try by running the following commands:

#### Shell

```
$ python days.py --weekday 2
Namespace(weekday=2)

$ python days.py --weekday 6
Namespace(weekday=6)

$ python days.py --weekday 9
usage: days.py [-h] [--weekday {1,2,3,4,5,6,7}]
days.py: error: argument --weekday: invalid choice: 9
        (choose from 1, 2, 3, 4, 5, 6, 7)
```

The first two examples work correctly because the input number is in the allowed range of values. However, if the input number is outside the defined range, like in the last example, then your app fails, displaying usage and error messages.



[Become a Python Expert »](#)

 Remove ads

## Providing and Customizing Help Messages in Arguments and Options

As you already know, a great feature of argparse is that it generates automatic usage and help messages for your applications. You can access these messages using the `-h` or `--help`

flag, which is included by default in any argparse CLI.

Up to this point, you've learned how to provide description and epilog messages for your apps. In this section, you'll continue improving your app's help and usage messages by providing enhanced messages for individual command-line arguments and options. To do this, you'll use the `help` and `metavar` arguments of `.add_argument()`.

Go back to your custom `ls` command and run the script with the `-h` switch to check its current output:

#### Shell

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

List the content of a directory

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

general output:
  path

detailed output:
  -l, --long

Thanks for using ls! :)
```

This output looks nice, and it's a good example of how argparse saves you a lot of work by providing usage and help message out of the box.

Note that only the `-h` or `--help` option shows a descriptive help message. In contrast, your own arguments `path` and `-l` or `--long` don't show a help message. To fix that, you can use the `help` argument.

Open your `ls.py` and update it like in the following code:

#### Python

```
# ls.py v8

import argparse
import datetime
from pathlib import Path

# ...

general = parser.add_argument_group("general output")
general.add_argument(
    "path",
    nargs="?",
    default=".",
    help="take the path to the target directory (default: %(default)s)",
)

detailed = parser.add_argument_group("detailed output")
detailed.add_argument(
    "-l",
    "--long",
    action="store_true",
    help="display detailed directory content",
)

# ...
```

In this update to `ls.py`, you use the `help` argument of `.add_argument()` to provide specific help messages for your arguments and options.

**Note:** As you already know, help messages support format specifiers like `%(prog)s`. You can use most of the arguments to `add_argument()` as format specifiers. For example, `%(default)s`, `%(type)s`, and so on.



Now go ahead and run the app with the `-h` flag again:

#### Shell

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

List the content of a directory

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

general output:
  path        take the path to the target directory (default: .)

detailed output:
  -l, --long  display detailed directory content

Thanks for using ls! :)
```

Now both `path` and `-l` show descriptive help messages when you run the app with the `-h` flag. Note that `path` includes its default value in its help message, which provides valuable information to your users.

Another desired feature is to have a nice and readable usage message in your CLI apps. The default usage message of `argparse` is pretty good already. However, you can use the `metavar` argument of `.add_argument()` to slightly improve it.

The `metavar` argument comes in handy when a command-line argument or option accepts input values. It allows you to give this input value a descriptive name that the parser can use to generate the help message.

As an example of when to use `metavar`, go back to your `point.py` example:

## Python

```
# point.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument("--coordinates", nargs=2)

args = parser.parse_args()

print(args)
```

If you run this application from your command line with the `-h` switch, then you get an output that'll look like the following:

## Shell

```
$ python point.py -h
usage: point.py [-h] [--coordinates COORDINATES COORDINATES]

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

By default, `argparse` uses the original name of command-line options to designate their corresponding input values in the usage and help messages, as you can see in the highlighted lines. In this specific example, the name `COORDINATES` in the plural may be confusing. Should your users provide the point's coordinates two times?

You can remove this ambiguity by using the metavar argument:

## Python

```
# point.py

import argparse

parser = argparse.ArgumentParser()

parser.add_argument(
    "--coordinates",
    nargs=2,
    metavar=("X", "Y"),
    help="take the Cartesian coordinates %(metavar)s",
)

args = parser.parse_args()

print(args)
```

In this example, you use a tuple as the value to `metavar`. The tuple contains the two coordinate names that people commonly use to designate a pair of Cartesian coordinates. You also provide a custom help message for `--coordinates`, including a format specifier with the `metavar` argument.

If you run the script with the `-h` flag, then you get the following output:

## Shell

```
$ python coordinates.py -h
usage: coordinates.py [-h] [--coordinates X Y]

options:
  -h, --help            show this help message and exit
  --coordinates X Y    take the Cartesian coordinates ('X', 'Y')
```

Now your app's usage and help messages are way clearer than before. Now your users will immediately know that they need to provide two numeric values, x and y, for the --coordinates option to work correctly.

**Learn Python Programming, By Example**

realpython.com



 Remove ads

## Defining Mutually Exclusive Argument and Option Groups

Another interesting feature that you can incorporate into your argparse CLIs is the ability to create mutually exclusive groups of arguments and options. This feature comes in handy when you have arguments or options that can't coexist in the same command construct.

Consider the following CLI app, which has --verbose and --silent options that can't coexist in the same command call:

Python

```
# groups.py

import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group(required=True)

group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-s", "--silent", action="store_true")

args = parser.parse_args()

print(args)
```

Having mutually exclusive groups for `--verbose` and `--silent` makes it impossible to use both options in the same command call:

#### Shell

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

You can't specify the `-v` and `-s` flags in the same command call. If you try to do it, then you get an error telling you that both options aren't allowed at the same time.

Note that the app's usage message showcases that `-v` and `-s` are mutually exclusive by using the pipe symbol (`|`) to separate them. This way of presenting the options must be interpreted as *use `-v` or `-s`, but not both*.

## Adding Subcommands to Your CLIs

Some command-line applications take advantage of subcommands to provide new features and functionalities. Applications like [pip](#), [pyenv](#), [Poetry](#), and [git](#), which are pretty popular among Python developers, make extensive use of subcommands.

For example, if you run `pip` with the `--help` switch, then you'll get the app's usage and help message, which includes the complete list of subcommands:

## Shell

```
$ pip --help
```

```
Usage:
```

```
  pip <command> [options]
```

```
Commands:
```

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.
...	

To use one of these subcommands, you just need to list it after the app's name. For example, the following command will list all the packages you've installed in your current Python environment:

## Shell

```
$ pip list
```

```
Package      Version
```

```
-----
```

```
pip          x.y.z
```

```
setuptools  x.y.z
```

```
...
```

Providing subcommands in your CLI applications is quite a useful feature. Fortunately, `argparse` also provides the required tool to implement this feature. If you want to arm your command-line apps with subcommands, then you can use the `.add_subparsers()` method of `ArgumentParser`.

As an example of using `.add_subparsers()`, say you want to create a CLI app to perform basic arithmetic operations, including addition, subtraction, multiplication, and division. You

want to implement these operations as subcommands in your app's CLI.

To build this app, you start by coding the app's core functionality, or the arithmetic operations themselves. Then you add the corresponding arguments to the app's CLI:

#### Python

```
1  # calc.py
2
3  import argparse
4
5  def add(a, b):
6      return a + b
7
8  def sub(a, b):
9      return a - b
10
11 def mul(a, b):
12     return a * b
13
14 def div(a, b):
15     return a / b
16
17 global_parser = argparse.ArgumentParser(prog="calc")
18 subparsers = global_parser.add_subparsers(
19     title="subcommands", help="arithmetic operations"
20 )
21
22 arg_template = {
23     "dest": "operands",
24     "type": float,
25     "nargs": 2,
26     "metavar": "OPERAND",
27     "help": "a numeric value",
28 }
29
30 add_parser = subparsers.add_parser("add", help="add two numbers a and b")
31 add_parser.add_argument(**arg_template)
```

```

32 add_parser.set_defaults(func=add)
33
34 sub_parser = subparsers.add_parser("sub", help="subtract two numbers a and
35 sub_parser.add_argument(**arg_template)
36 sub_parser.set_defaults(func=sub)
37
38 mul_parser = subparsers.add_parser("mul", help="multiply two numbers a and
39 mul_parser.add_argument(**arg_template)
40 mul_parser.set_defaults(func=mul)
41
42 div_parser = subparsers.add_parser("div", help="divide two numbers a and b'
43 div_parser.add_argument(**arg_template)
44 div_parser.set_defaults(func=div)
45
46 args = global_parser.parse_args()
47
48 print(args.func(*args.operands))

```

Here's a breakdown of how the code works:

- **Lines 5 to 15** define four functions that perform the basic arithmetic operations of addition, subtraction, multiplication, and division. These functions will provide the operations behind each of your app's subcommands.
- **Line 17** defines the command-line argument parser as usual.
- **Lines 18 to 20** define a subparser by calling `.add_subparsers()`. In this call, you provide a title and a help message.
- **Lines 22 to 28** define a template for your command-line arguments. This template is a dictionary containing sensitive values for the required arguments of `.add_argument()`. Each argument will be called operands and will consist of two floating-point values. Defining this template allows you to avoid repetitive code when creating the command-line arguments.
- **Line 30** adds a parser to the subparser object. The name of this subparser is add and



will represent your subcommand for addition operations. The `help` argument defines a help message for this parser in particular.

- **Line 31** adds the operands command-line argument to the `add` subparser using `.add_argument()` with the argument template. Note that you need to use the [dictionary unpacking operator](#) (`**`) to extract the argument template from `arg_template`.
- **Line 32** uses `.set_defaults()` to assign the `add()` callback function to the `add` subparser or subcommand.

Lines 34 to 44 perform actions similar to those in lines 30 to 32 for the rest of your three subcommands, `sub`, `mul`, and `div`. Finally, line 48 calls the `func` attribute from `args`. This attribute will automatically call the function associated with the subcommand at hand.

Go ahead and try out your new CLI calculator by running the following commands:

## Shell

```
$ python calc.py add 3 8
```

```
11.0
```

```
$ python calc.py sub 15 5
```

```
10.0
```

```
$ python calc.py mul 21 2
```

```
42.0
```

```
$ python calc.py div 12 2
```

```
6.0
```

```
$ python calc.py -h
```

```
usage: calc [-h] {add,sub,mul,div} ...
```

options:

-h, --help show this help message and exit

subcommands:

{add,sub,mul,div}	arithmetic operations
add	add two numbers a and b
sub	subtract two numbers a and b
mul	multiply two numbers a and b
div	divide two numbers a and b

```
$ python calc.py div -h
```

```
usage: calc div [-h] OPERAND OPERAND
```

positional arguments:

OPERAND a numeric value

options:

-h, --help show this help message and exit

Cool! All your subcommands work as expected. They take two numbers and perform the target arithmetic operation with them. Note that now you have usage and help messages for the app and for each subcommand too.

## Handling How Your CLI App's Execution Terminates

When creating CLI applications, you'll find situations in which you'll need to terminate the execution of an app because of an error or an exception. A common practice in this situation is to exit the app while emitting an [error code](#) or [exit status](#) so that other apps or the operating system can understand that the app has terminated because of an error in its execution.

Typically, if a command exits with a zero code, then it has succeeded. Meanwhile, a nonzero exit status indicates a failure. The drawback of this system is that while you have a single, well-defined way to indicate success, you have various ways to indicate failure, depending on the problem at hand.

Unfortunately, there's no definitive standard for error codes or exit statuses. Operating systems and programming languages use different styles, including decimal or [hexadecimal](#) numbers, alphanumeric codes, and even a phrase describing the error. Unix programs generally use 2 for command-line syntax errors and 1 for all other errors.

In Python, you'll commonly use integer values to specify the system exit status of a CLI app. If your code returns None, then the exit status is zero, which is considered a **successful termination**. Any nonzero value means **abnormal termination**. Most systems require the exit code to be in the range from 0 to 127, and produce undefined results otherwise.

When building CLI apps with argparse, you don't need to worry about returning exit codes for successful operations. However, you should return an appropriate exit code when your app abruptly terminates its execution due to an error other than command syntax errors, in

which case `argparse` does the work for you out of the box.

The `argparse` module, specifically the `ArgumentParser` class, has two dedicated methods for terminating an app when something isn't going well:

Method	Description
<code>.exit(status=0, message=None)</code>	Terminates the app, returning the specified status and printing message if given
<code>.error(message)</code>	Prints a usage message that incorporates the provided message and terminates the app with a status code of 2

Both methods print directly to the [standard error stream](#), which is dedicated to error reporting. The `.exit()` method is appropriate when you need complete control over which status code to return. On the other hand, the `.error()` method is internally used by `argparse` for command-line syntax errors, but you can use it whenever it's necessary and appropriate.

As an example of when to use these methods, consider the following update to your custom `ls` command:

## Python

```
# ls.py v9

import argparse
import datetime
from pathlib import Path

# ...

target_dir = Path(args.path)

if not target_dir.exists():
    parser.exit(1, message="The target directory doesn't exist")

# ...
```

In the conditional statement that checks if the target directory exists, instead of using `raise SystemExit(1)`, you use `ArgumentParser.exit()`. This makes your code more focused on the selected tech stack, which is the `argparse` framework.

To check how your app behaves now, go ahead and run the following commands:

 Windows

  Linux + macOS

## Windows PowerShell

```
PS> python ls.py .\non_existing\
The target directory doesn't exist

PS> echo $LASTEXITCODE
1
```

The app terminates its execution immediately when the target directory doesn't exist. If

you're on a Unix-like system, such as Linux or macOS, then you can inspect the `$?` shell

variable to confirm that your app has returned 1 to signal an error in its execution. If you're on Windows, then you can check the contents of the `$LASTEXITCODE` variable.

Providing consistent status codes in your CLI applications is a best practice that'll allow you and your users to successfully integrate your app in their shell scripting and command pipes.

## Conclusion

Now you know what a command-line interface is and what its main components are, including arguments, options, and subcommands. You also learned how to create fully functional **CLI applications** using the **argparse** module from the Python standard library.

**In this tutorial, you've learned how to:**

- Get started with **command-line interfaces**
- **Organize** and **lay out** a command-line project in Python
- Use Python's **argparse** to create **command-line interfaces**
- Customize most aspects of a CLI with some **powerful features** of **argparse**

Knowing how to write effective and intuitive command-line interfaces is a great skill to have as a developer. Writing good CLIs for your apps allows you to give your users a pleasant user experience while interacting with your applications.

**Source Code:** [Click here to download the source code](#) that you'll use to build command-line interfaces with **argparse**.

Mark as Completed





This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Building Command Line Interfaces With argparse](#)



## Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Send Me Python Tricks »

## About **Leodanis Pozo Ramos**

Leodanis is an industrial engineer who loves Python and software development. He's a self-taught Python developer with 6+ years of experience. He's an avid technical writer with a growing number of articles published on Real Python and other sites.

» [More aboutLeodanis](#)

*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

Davide

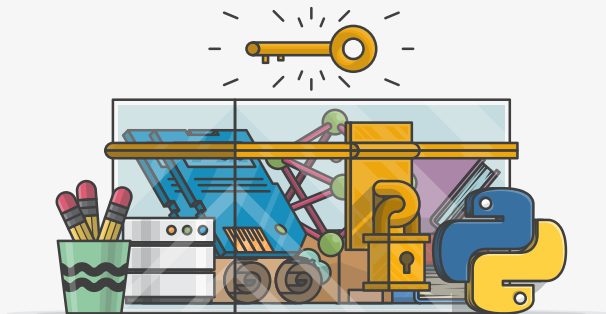
Geir Arne

Ian

Joanna

Kate

Master Real-World Python Skills  
With Unlimited Access to Real Python





Join us and get access to thousands of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

## What Do You Think?

Rate this article:



Tweet



Share



Share



Email

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.

**Commenting Tips:** The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next [“Office Hours” Live Q&A Session](#). Happy Pythoning!

## Keep Learning

Related Tutorial Categories: [intermediate](#) [python](#)

Recommended Video Course: [Building Command Line Interfaces With argparse](#)

### Python Tricks The Book

A Buffet of Awesome Python Features

[Get Your Free Sample Chapter](#)



[i Remove ads](#)

© 2012–2022 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) · [Instagram](#) ·  
[Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)

❤ Happy Pythoning!