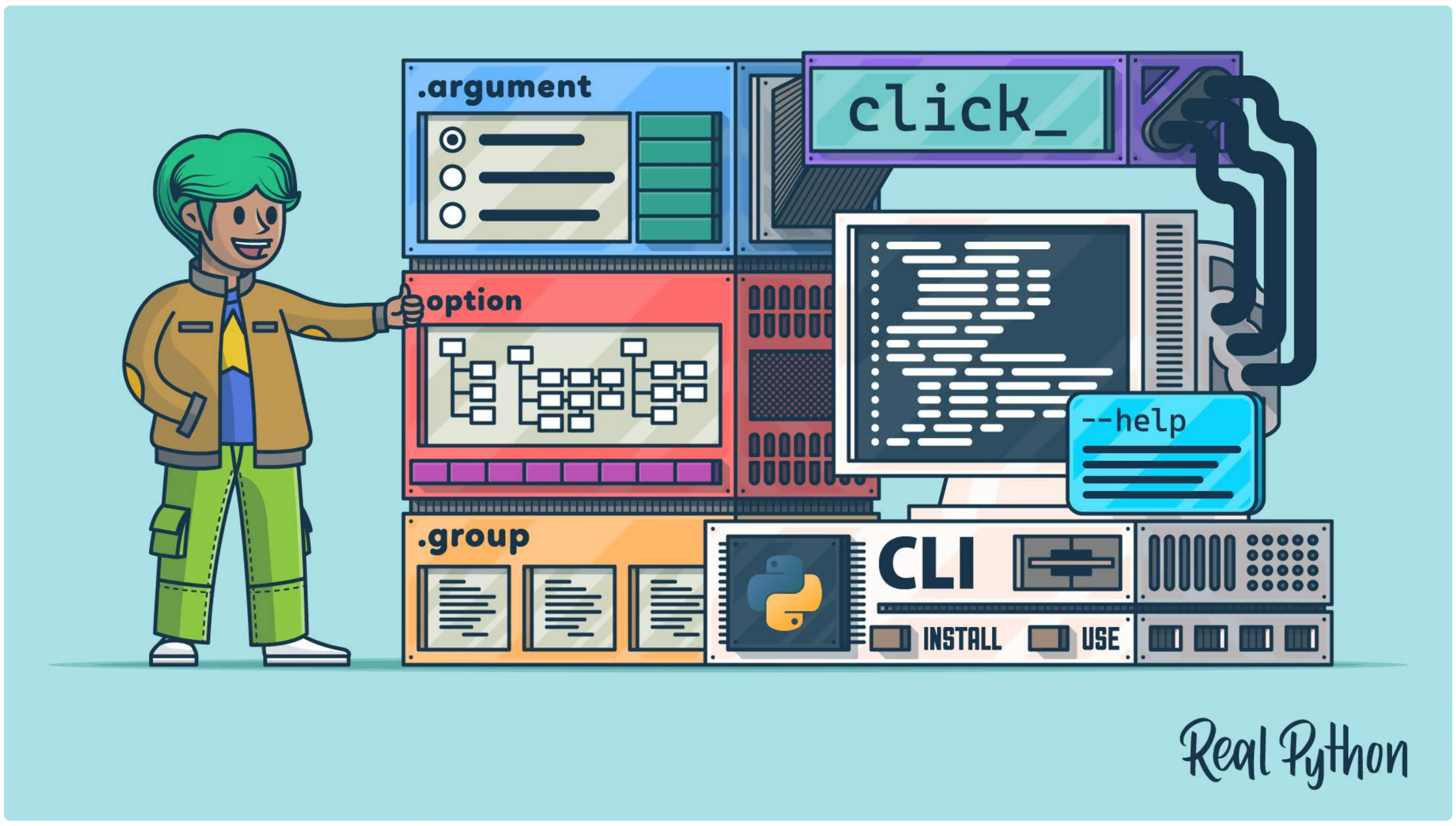

Click and Python: Build Extensible and Composable CLI Apps

by Leodanis Pozo Ramos Aug 23, 2023 0 Comments

devops intermediate python tools

Mark as Completed

Table of Contents

Help

You can use the **Click** library to quickly provide your Python automation and tooling scripts with an extensible, composable, and user-friendly command-line interface (CLI). Whether you're a developer, data scientist, DevOps engineer, or someone who often uses Python to automate repetitive tasks, you'll very much appreciate Click and its unique features.

In the Python ecosystem, you'll find multiple libraries for creating CLIs, including `argparse` from the standard library, Typer, and a few others. However, Click offers a robust, mature, intuitive, and feature-rich solution.

In this tutorial, you'll learn how to:

- Create **command-line interfaces** with **Click** and Python
- Add **arguments**, **options**, and **subcommands** to your CLI apps
- Enhance the **usage** and **help pages** of your CLI apps with Click
- Prepare a Click CLI app for **installation**, **use**, and **distribution**

To get the most out of this tutorial, you should have a good understanding of Python programming, including topics such as using decorators. It'll also be helpful if you're familiar with using your current operating system's command line or terminal.

Get Your Code: Click here to download the sample code that you'll use to build your CLI app with Click and Python.

Creating Command-Line Interfaces With Click and Python

The Click library enables you to quickly create robust, feature-rich, and extensible command-line interfaces (CLIs) for your scripts and tools. This library can significantly speed up your development process because it allows you to focus on the application's logic and leave CLI creation and management to the library itself.

Click is a great alternative to the `argparse` module, which is the default CLI framework in the Python standard library. Next up, you'll learn what sets it apart.

Why Use Click for CLI Development

Compared with `argparse`, Click provides a more flexible and intuitive framework for creating CLI apps that are highly extensible. It allows you to gradually compose your apps without restrictions and with a minimal amount of code. This code will be readable even when your CLI grows and becomes more complex.

Click's application programming interface (API) is highly intuitive and consistent. The API takes advantage of Python decorators, allowing you to add arguments, options, and subcommands to your CLIs quickly.

Functions are fundamental in Click-based CLIs. You have to write functions that you can then wrap with the appropriate decorators to create arguments, commands, and so on.

Click has several desirable features that you can take advantage of. For example, Click apps:

- Can be **lazily composable** without restrictions
- Follow the Unix **command-line conventions**
- Support loading values from **environment variables**
- Support custom **prompts** for input values
- Handle **paths** and **files** out of the box
- Allow arbitrary nesting of commands, also known as **subcommands**

You'll find that Click has many other cool features. For example, Click keeps information about all of your arguments, options, and commands. This way, it can generate usage and help pages for the CLI, which improves the user experience.

When it comes to processing user input, Click has a strong understanding of data types. Because of this feature, the library generates consistent error messages when the user provides the wrong type of input.

Now that you have a general understanding of Click's most relevant features, it's time to get your hands dirty and write your first Click app.

How to Install and Set Up Click: Your First CLI App

Unlike `argparse`, Click doesn't come in the Python standard library. This means that you need to install Click as a dependency of your CLI project to use the library. You can install Click from PyPI using `pip`. First, you should create a Python virtual environment to work on. You can do all of that with the following platform-specific commands:

Windows Linux + macOS

Windows Command Prompt

```
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install click
```

With the first two commands, you create and activate a Python virtual environment called `venv` in your working directory. Once the environment is active, you install Click using `pip`.

Great! You've installed Click in a fresh virtual environment. Now go ahead and fire up your favorite code editor. Create a new `hello.py` file and add the following content to it:

Python

```python
# hello.py

import click

@click.command("hello")
@click.version_option("0.1.0", prog_name="hello")
def hello():
    click.echo("Hello, World!")

if __name__ == "__main__":
    hello()
```

In this file, you first import the `click` package. Then you create a function called `hello()`. In this function, you print a message to the screen. To do this, you use the Click `echo()` function instead of your old friend `print()`. Why would you do that?

The `echo()` function applies some error corrections in case the terminal program has configuration issues. It also supports colors and other styles in the output. It automatically removes any styling if the output stream is a file rather than the standard output. So, when working with Click, you should use `echo()` to handle the app's output.

You use two decorators on top of this function. The `@click.command` decorator declares `hello()` as a Click command with the name `"hello"`. The `@click.version_option` decorator sets the CLI app's version and name. This information will show up when you run the app with the `--version` command-line option.

Finally, you add the name-main idiom to call the `hello()` function when you run the file as an executable program.

Go ahead and run the app from your command line:

Shell

```
(venv) $ python hello.py
Hello, World!

(venv) $ python hello.py --version
hello, version 0.1.0

(venv) $ python hello.py --help
Usage: hello.py [OPTIONS]

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.
```

When you run the script without arguments, you get `Hello, World!` displayed on your screen. If you use the `--version` option, then you get information about the app's name and version. Note that Click automatically provides the `--help` option, which you can use to access the app's main help page.

That was cool! With a few lines of code and the power of Click, you've created your first CLI app. It's a minimal app, but it's enough to get a grasp of what you'll be capable of creating with Click. To continue your CLI journey, you'll learn how to make your apps take input arguments from the user.

Adding Arguments to a Click App

In CLI development, an **argument** is a required or optional piece of information that a command uses to perform its intended action. Commands typically accept arguments, which you can provide as a whitespace-separated or comma-separated list on your command line.

In this section, you'll learn how to take command-line arguments in your Click applications. You'll start with the most basic form of arguments and walk through different types of arguments, including paths, files, environment variables, and more.

Adding Basic Arguments

You can use the `@click.argument` decorator to make your Click app accept and parse arguments that you provide at the command line directly. Click parses the most basic arguments as strings that you can then pass to the underlying function.

To illustrate, say that you want to create a small CLI app to mimic the Unix `ls` command. In its most minimal variation, this command takes a directory as an argument and lists its content. If you're on Linux or macOS, then you can try out the command like in the following example:

Shell

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

This example assumes that you have a folder called `sample/` in your current working directory. Your folder contains the files `hello.txt`, `lorem.md`, and `realpython.md`, which are listed on the same output line.

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---          07/11/2023 10:06 AM             88 hello.txt
-a---          07/11/2023 10:06 AM           2629 lorem.md
-a---          07/11/2023 10:06 AM            429 realpython.md
```

The PowerShell `ls` command issues a table containing detailed information on every file and subdirectory in your target directory. So, in the upcoming versions of `ls.py`, you'll be mimicking the Unix version of this command instead of the PowerShell version.

To follow along with this tutorial and get the same outputs, you can download the companion sample code and resources, including the `sample/` directory, by clicking the link below:

Get Your Code: Click here to download the sample code that you'll use to build your CLI app with Click and Python.

How can you emulate this command behavior using Click and Python? You can do something like the following:

Python

```python
# ls.py v1

from pathlib import Path

import click

@click.command()
@click.argument("path")
def cli(path):
    target_dir = Path(path)
    if not target_dir.exists():
        click.echo("The target directory doesn't exist")
        raise SystemExit(1)

    for entry in target_dir.iterdir():
        click.echo(f"{entry.name:{len(entry.name) + 5}}", nl=False)

    click.echo()

if __name__ == "__main__":
    cli()
```

This is your first version of the `ls` command emulator app. You start by importing the `Path` class from `pathlib`. You'll use this class to efficiently manage paths in your application. Next, you import `click` as usual.

Your `ls` emulator needs a single function to perform its intended task. You call this function `cli()`, which is a common practice. Click apps typically name the entry-point command `cli()`, as you'll see throughout this tutorial.

In this example, you use the `@click.command` decorator to define a command. Then you use the `@click.argument` decorator with the string `"path"` as an argument. This call to the decorator adds a new command-line argument called `"path"` to your custom `ls` command.

Note that the name of the command-line argument must be the same as the argument to `cli()`. This way, you're passing the user input directly to your processing code.

Inside `cli()`, you create a new `Path` instance using the user input. Then you check the input path. If the path doesn't exist, then you inform the user and exit the app with an appropriate exit status. If the path exists, then the `for` loop lists the directory content, simulating what the Unix `ls` command does.

The call to `click.echo()` at the end of `cli()` allows you to add a new line at the end of the output to match the `ls` behavior.

Note: To dive deeper into listing the content of a directory, check out How to Get a List of All Files in a Directory With Python.

If you run the commands below, then you'll get the following results:

Shell

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

(venv) $ python ls.py non_existing/
The target directory doesn't exist

(venv) $ python ls.py
Usage: ls.py [OPTIONS] PATH
Try 'ls.py --help' for help.

Error: Missing argument 'PATH'.
```

If you run the app with a valid directory path, then you get the directory content listed. If the target directory doesn't exist, then you get an informative message. Finally, running the app without an argument causes the app to fail, displaying the help page.

Note: You may find that the order in which `ls.py` lists the files doesn't match the order in your output or in the original `ls` command's output. That's because the `.iterdir()` method yields directory entries in arbitrary order.

How does that look for not even twenty lines of Python code? Great! However, Click offers you a better way to do this. You can take advantage of Click's power to automatically handle file paths in your applications.

Using Path Arguments

The `@click.argument` decorator accepts an argument called `type` that you can use to define the target data type of the argument at hand. In addition to this, Click provides a rich set of custom classes that allow you to consistently handle different data types, including paths.

Note: You'll learn more about Click's custom parameter types throughout this tutorial, especially in the Providing Parameter Types for Arguments and Options section.

In the example below, you rewrite the `ls` app using Click's capabilities:

Python

```python
# ls.py v2

from pathlib import Path

import click

@click.command()
@click.argument(
    "path",
    type=click.Path(
        exists=True,
        file_okay=False,
        readable=True,
        path_type=Path,
    ),
)
def cli(path):
    for entry in path.iterdir():
        click.echo(f"{entry.name:{len(entry.name) + 5}}", nl=False)

    click.echo()

if __name__ == "__main__":
    cli()
```

In this new version of `ls.py`, you pass a `click.Path` object to the `type` argument of `@click.argument`. With this addition, Click will treat any input as a path object.

To instantiate the `click.Path()` class in this example, you use several arguments:

- **`exists`**: If you set it to `True`, then Click will make sure that the path exists.
- **`file_okay`**: If you set it to `False`, then Click will make sure that the input path doesn't point to a file.
- **`readable`**: If you set it to `True`, then Click will make sure that you can read the content of the target directory.
- **`path_type`**: If you set it to `pathlib.Path`, then Click will turn the input into a `Path` object.

With these settings in place, your `cli()` function is more concise. It only needs the `for` loop to list the directory content. Go ahead and run the following commands to test the new version of your `ls.py` script:

Shell

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

(venv) $ python ls.py non_existing/
Usage: ls.py [OPTIONS] PATH
Try 'ls.py --help' for help.

Error: Invalid value for 'PATH': Directory 'non_existing/' does not exist.
```

Again, when you run the app with a valid directory path, you get the directory content listed. If the target directory doesn't exist, then Click handles the issue for you. You get a nice usage message and an error message describing the current issue. That's an even better behavior if you compare it with your first `ls.py` version.

Accepting Variadic Arguments

In Click's terminology, a **variadic argument** is one that accepts an undetermined number of input values at the command line. This type of argument is pretty common in CLI development. For example, the Unix `ls` command takes advantage of this feature, allowing you to process multiple directories at a time.

To give it a try, make a copy of your `sample/` folder and run the following command:

Shell

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

sample_copy/:
hello.txt     lorem.md      realpython.md
```

The `ls` command can take multiple target directories at the command line. The output will list the content of each directory, as you can conclude from the example above. How can you emulate this behavior using Click and Python?

The `@click.argument` decorator takes an argument called `nargs` that allows you to predefine the number of values that an argument can accept at the command line. If you set `nargs` to `-1`, then the underlying argument will collect an undetermined number of input values in a tuple.

Here's how you can take advantage of `nargs` to accept multiple directories in your `ls` emulator:

Python

```python
# ls.py v3

from pathlib import Path

import click

@click.command()
@click.argument(
    "paths",
    nargs=-1,
    type=click.Path(
        exists=True,
        file_okay=False,
        readable=True,
        path_type=Path,
    ),
)
def cli(paths):
    for i, path in enumerate(paths):
        if len(paths) > 1:
            click.echo(f"{path}/:")
        for entry in path.iterdir():
            click.echo(f"{entry.name:{len(entry.name) + 5}}", nl=False)
        if i < len(paths) - 1:
            click.echo("\n")
        else:
            click.echo()

if __name__ == "__main__":
    cli()
```

In the first highlighted line, you change the argument's name from `"path"` to `"paths"` because now the argument will accept multiple directory paths. Then you set `nargs` to `-1` to indicate that this argument will accept multiple values at the command line.

In the `cli()` function, you change the argument's name to match the command-line argument's name. Then you start a loop over the input paths. The conditional statement prints the name of the current directory, simulating what the original `ls` command does.

Then you run the usual loop to list the directory content, and finally, you call `echo()` to add a new blank line after the content of each directory. Note that you use the `enumerate()` function to get an index for every path. This index allows you to figure out when the output should end so that you can skip the extra blank line and mimic the `ls` behavior.

With these updates in place, you can run the app again:

Shell

```shell
(venv) $ python ls.py sample/ sample_copy/
sample/:
lorem.md     realpython.md     hello.txt

sample_copy/:
lorem.md     realpython.md     hello.txt
```

Now your custom `ls` command behaves similarly to the original Unix `ls` command when you pass multiple target directories at the command line. That's great! You've learned how to implement variadic arguments with Click.

Taking File Arguments

Click provides a parameter type called `File` that you can use when the input for a given command-line argument must be a file. With `File`, you can declare that a given parameter is a file. You can also declare whether your app should open the file for reading or writing.

To illustrate how you can use the `File` parameter type, say that you want to emulate the basic functionality of the Unix cat command. This command reads files sequentially and writes their content to the standard output, which is your screen:

Shell

```
$ cat sample/hello.txt sample/realpython.md
Hello, Pythonista!

Welcome to Real Python!
At Real Python you'll learn all things Python from the ground up.
Their tutorials, books, and video courses are created, curated,
and vetted by a community of expert Pythonistas. With new content
published weekly, custom Python learning paths, and interactive
code challenges, you'll always find something to boost your skills.
Join 3,000,000+ monthly readers and take your Python skills to the
next level at realpython.com.
```

In this example, you use `cat` to concatenate the content of two files from your sample directory. The following app mimics this behavior using Click's `File` parameter type:

Python

```python
# cat.py

import click

@click.command()
@click.argument(
    "files",
    nargs=-1,
    type=click.File(mode="r"),
)
def cli(files):
    for file in files:
        click.echo(file.read().rstrip())

if __name__ == "__main__":
    cli()
```

In this example, you set the `type` argument to `click.File`. The `"r"` mode means that you're opening the file for reading.

Inside `cli()`, you start a `for` loop to iterate over the input files and print their content to the screen. It's important to note that you don't need to worry about closing each file once you've read its content. The `File` type automatically closes it for you once the command finishes running.

Here's how this app works in practice:

Shell

```
(venv) $ python cat.py sample/hello.txt sample/realpython.md
Hello, Pythonista!

Welcome to Real Python!
At Real Python you'll learn all things Python from the ground up.
Their tutorials, books, and video courses are created, curated,
and vetted by a community of expert Pythonistas. With new content
published weekly, custom Python learning paths, and interactive
code challenges, you'll always find something to boost your skills.
Join 3,000,000+ monthly readers and take your Python skills to the
next level at realpython.com.
```

Your `cat.py` script works pretty similarly to the Unix `cat` command. It accepts multiple files at the command line, opens them for reading, reads their content, and prints it to the screen sequentially. Great job!

Providing Options in Your Click Apps

Command options are another powerful feature of Click applications. Options are named, non-required arguments that modify a command's behavior. You pass an option to a command using a specific name, which typically has a prefix of one dash (`-`) or two dashes (`--`) on Unix systems. On Windows, you may also find options with other prefixes, such as a slash (`/`).

Because options have names, they enhance the usability of a CLI app. In Click, options can do the same as arguments. Additionally, options have a few extra features. For example, options can:

- Prompt for input values
- Act as flags or feature switches
- Pull their value from environment variables

Unlike arguments, options can only accept a fixed number of input values, and this number defaults to 1. Additionally, you can specify an option multiple times using multiple options, but you can't do this with arguments.

In the following sections, you'll learn how to add options to your Click command and how options can help you improve your users' experience while they work with your CLI apps.

Adding Single-Value Options

To add an option to a Click command, you'll use the `@click.option` decorator. The first argument to this decorator will hold the option's name.

CLI options often have a long and a short name. The long name typically describes what the option does, while the short name is commonly a single-letter shortcut. To Click, names with a single leading dash are short names, while names with two leading dashes are long ones.

In Click, the most basic type of option is a single-value option, which accepts one argument at the command line. If you don't provide a parameter type for the option value, then Click assumes the `click.STRING` type.

To illustrate how you can create options with Click, say that you want to write a CLI app that emulates the Unix `tail` command. This command displays the tail end of a text file:

Shell

```shell
$ tail sample/lorem.md
ac. Nulla sapien nulla, egestas at pretium ac, feugiat nec arcu. Donec
ullamcorper laoreet odio, id posuere nisl ullamcorper at.

### Nam Aliquam Ultricies Pharetra

Nam aliquam ultricies pharetra. Pellentesque accumsan finibus ex porta
aliquet. Morbi placerat sagittis tortor, ut maximus sem iaculis sit amet.
Aliquam sit amet libero dapibus, vehicula arcu non, pulvinar felis.
Suspendisse a risus magna. Nulla facilisi. Donec eu consequat ligula, iaculis
aliquet augue.

$ tail --lines 3 sample/lorem.md
Aliquam sit amet libero dapibus, vehicula arcu non, pulvinar felis.
Suspendisse a risus magna. Nulla facilisi. Donec eu consequat ligula, iaculis
aliquet augue.
```

By default, `tail` displays the last ten lines of the input file. However, the command has an `-n` or `--lines` option that allows you to tweak that number, as you can see in the second execution above, where you only printed the last three lines of `lorem.md`.

You can use Click to emulate the behavior of `tail`. In this case, you need to add an option using the `@click.option` decorator as in the code below:

Python

```python
# tail.py

from collections import deque

import click

@click.command()
@click.option("-n", "--lines", type=click.INT, default=10)
@click.argument(
    "file",
    type=click.File(mode="r"),
)
def cli(file, lines):
    for line in deque(file, maxlen=lines):
        click.echo(line, nl=False)

if __name__ == "__main__":
    cli()
```

In this example, you first import the deque data type from the collections module. You'll use this type to quickly get the final lines of your input file. Then you import click as usual.

Note: Your use of deque for this example comes from the Python documentation on deque. Check out the section on deque recipes for further examples.

In the highlighted line, you call the @click.option decorator to add a new option to your Click command. The first two arguments in this call provide short and long names for the option, respectively.

Because the user input must be an integer number, you use click.INT to define the parameter's type. The default behavior of tail is to display the final ten lines, so you set default to 10 and discover another cool feature of Click's options. They can have default values.

Next, you add an argument called "file", which is of type click.File. You already know how the File type works.

In cli(), you take the file and the number of lines as arguments. Then you loop over the last lines using a deque object. This specific deque object can only store up to lines items. This guarantees that you get the desired number of lines from the end of the input file.

Go ahead and give tail.py a try by running the following commands:

Shell

```
(venv) $ python tail.py sample/lorem.md
ac. Nulla sapien nulla, egestas at pretium ac, feugiat nec arcu. Donec
ullamcorper laoreet odio, id posuere nisl ullamcorper at.

### Nam Aliquam Ultricies Pharetra

Nam aliquam ultricies pharetra. Pellentesque accumsan finibus ex porta
aliquet. Morbi placerat sagittis tortor, ut maximus sem iaculis sit amet.
Aliquam sit amet libero dapibus, vehicula arcu non, pulvinar felis.
Suspendisse a risus magna. Nulla facilisi. Donec eu consequat ligula, iaculis
aliquet augue.

(venv) $ python tail.py --lines 3 sample/lorem.md
Aliquam sit amet libero dapibus, vehicula arcu non, pulvinar felis.
Suspendisse a risus magna. Nulla facilisi. Donec eu consequat ligula, iaculis
aliquet augue.

(venv) $ python tail.py --help
Usage: tail.py [OPTIONS] FILE

Options:
  -n, --lines INTEGER
  --help               Show this message and exit.
```

Your custom `tail` command works similarly to the original Unix `tail` command. It takes a file and displays the last ten lines by default. If you provide a different number of lines with the `--lines` option, then the command displays only your desired lines from the end of the input file.

When you check the help page of your `tail` command, you see that the `-n` or `--lines` option now shows up under the `Options` heading. By default, you also get information about the option's parameter type, which is an integer number in this example.

Creating Multi-Value Options

Sometimes, you need to implement an option that takes more than one input value at the command line. Unlike arguments, Click options only support a fixed number of input values. You can configure this number using the `nargs` argument of `@click.option`.

The example below accepts a `--size` option that needs two input values, `width` and `height`:

Python

```python
# rectangle.py v1

import click

@click.command()
@click.option("--size", nargs=2, type=click.INT)
def cli(size):
    width, height = size
    click.echo(f"size: {size}")
    click.echo(f"{width} × {height}")

if __name__ == "__main__":
    cli()
```

In this example, you set `nargs` to `2` in the call to the `@click.option` decorator that defines the `--size` option. This setting tells Click that the option will accept two values at the command line.

Here's how this toy app works in practice:

Shell

```shell
(venv) $ python rectangle.py --size 400 200
size: (400, 200)
400 × 200

(venv) $ python rectangle.py --size 400
Error: Option '--size' requires 2 arguments.

(venv) $ python rectangle.py --size 400 200 100
Usage: rectangle.py [OPTIONS]
Try 'rectangle.py --help' for help.

Error: Got unexpected extra argument (100)
```

The `--size` option accepts two input values at the command line. Click stores these values in a tuple that you can process inside the `cli()` function. Note how the `--size` option doesn't accept fewer or more than two input values.

Click provides an alternative way to create multi-value options. Instead of using the `nargs` argument of `@click.option`, you can set the `type` argument to a tuple. Consider the following alternative implementation of your `rectangle.py` script:

Python

```python
# rectangle.py v2

import click

@click.command()
@click.option("--size", type=(click.INT, click.INT))
def cli(size):
    width, height = size
    click.echo(f"size: {size}")
    click.echo(f"{width} × {height}")

if __name__ == "__main__":
    cli()
```

In this alternative implementation of `rectangle.py`, you set the `type` argument to a tuple of integer values. Note that you can also use the `click.Tuple` parameter type to get the same result. Using this type will be more explicit, and you only have to do `type=click.Tuple([int, int])`.

Go ahead and try out this new variation of your app:

Shell

```
(venv) $ python rectangle.py --size 400 200
size: (400, 200)
400 × 200

(venv) $ python rectangle.py --size 400
Error: Option '--size' requires 2 arguments.

(venv) $ python rectangle.py --size 400 200 100
Usage: rectangle.py [OPTIONS]
Try 'rectangle.py --help' for help.

Error: Got unexpected extra argument (100)
```

This implementation works the same as the one that uses `nargs=2`. However, by using a tuple for the `type` argument, you can customize the parameter type of each item in the tuple, which can be a pretty handy feature in some situations.

To illustrate how `click.Tuple` can help you, consider the following example:

Python

```python
# person.py

import click

@click.command()
@click.option("--profile", type=click.Tuple([str, int]))
def cli(profile):
    click.echo(f"Hello, {profile[0]}! You're {profile[1]} years old!")

if __name__ == "__main__":
    cli()
```

In this example, the `--profile` option takes a two-item tuple. The first item should be a string representing a person's name. The second item should be an integer representing their age.

Here's how this toy app works in practice:

Shell

```
(venv) $ python person.py --profile John 35
Hello, John! You're 35 years old!

(venv) $ python person.py --profile Jane 28.5
Usage: person.py [OPTIONS]
Try 'person.py --help' for help.

Error: Invalid value for '--profile': '28.5' is not a valid integer.
```

The `--profile` option accepts a string and an integer value. If you use a different data type, then you'll get an error. Click does the type validation for you.

Specifying an Option Multiple Times

Repeating an option multiple times at the command line is another cool feature that you can implement in your CLI apps with Click. As an example, consider the following toy app, which takes a `--name` option and displays a greeting. The app allows you to specify `--name` multiple times:

Python

```python
# greet.py

import click

@click.command()
@click.option("--name", multiple=True)
def cli(name):
    for n in name:
        click.echo(f"Hello, {n}!")

if __name__ == "__main__":
    cli()
```

The `multiple` argument to `@click.option` is a Boolean flag. If you set it to `True`, then you can specify the underlying option multiple times.

Here's how this app works in practice:

Shell

```
(venv) $ python greet.py --name Pythonista --name World
Hello, Pythonista!
Hello, World!
```

In this command, you specify the `--name` option two times. Each time, you use a different input value. As a result, the application prints two greetings to your screen, one greeting per option repetition. Next up, you'll learn more about Boolean flags.

Defining Options as Boolean Flags

Boolean flags are options that you can enable or disable. Click accepts two types of Boolean flags. The first type allows you to define *on* and *off* switches. The second type only provides an *on* switch. To define a flag with *on* and *off* switches, you can provide the two flags separated by a slash (`/`).

As an example of an *on* and *off* flag, consider the following app:

Python

```python
# upper_greet.py

import click

@click.command()
@click.argument("name", default="World")
@click.option("--upper/--no-upper", default=False)
def cli(name, upper):
    message = f"Hello, {name}!"
    if upper:
        message = message.upper()
    click.echo(message)

if __name__ == "__main__":
    cli()
```

In the highlighted line, you define an option that works as an *on* and *off* flag. In this example, `--upper` is the *on* (or `True`) switch, while `--no-upper` is the *off* (or `False`) switch. Note that the *off* flag doesn't have to use the `no-` prefix. You can name it what you want, depending on your specific use case.

Then you pass `upper` as an argument to your `cli()` function. If `upper` is true, then you uppercase the greeting message. Otherwise, the message keeps its original casing. Note that the default value for this flag is `False`, which means that the app will display the message without changing its original casing.

Here's how this app works in practice:

Shell

```
(venv) $ python upper_greet.py Pythonista --upper
HELLO, PYTHONISTA!

(venv) $ python upper_greet.py Pythonista --no-upper
Hello, Pythonista!

(venv) $ python upper_greet.py Pythonista
Hello, Pythonista!
```

When you run your app with the `--upper` flag, you get the greeting in uppercase. When you run the app with the `--no-upper` flag, you get the message in its original casing. Finally, running the app without a flag displays the message without modification because the default value for the flag is `False`.

The second type of Boolean flag only provides an *on*, or `True`, switch. In this case, if you provide the flag at the command line, then its value will be `True`. Otherwise, its value will be `False`. You can set the `is_flag` argument of `@click.option()` to `True` when you need to create this type of flag.

To illustrate how you can use these flags, get back to your `ls` simulator. This time, you'll add an `-l` or `--long` flag that mimics the behavior of the equivalent flag in the original Unix `ls` command. Here's the updated code:

Python

```python
# ls.py v4

from datetime import datetime
from pathlib import Path

import click

@click.command()
@click.option("-l", "--long", is_flag=True)
@click.argument(
    "paths",
    nargs=-1,
    type=click.Path(
        exists=True,
        file_okay=False,
        readable=True,
        path_type=Path,
    ),
)
def cli(paths, long):
    for i, path in enumerate(paths):
        if len(paths) > 1:
            click.echo(f"{path}/:")
        for entry in path.iterdir():
            entry_output = build_output(entry, long)
            click.echo(f"{entry_output:{len(entry_output) + 5}}", nl=long)
        if i < len(paths) - 1:
            click.echo("" if long else "\n")
        elif not long:
            click.echo()

def build_output(entry, long=False):
    if long:
        size = entry.stat().st_size
        date = datetime.fromtimestamp(entry.stat().st_mtime)
        return f"{size:>6d} {date:%b %d %H:%M:%S} {entry.name}"
    return entry.name

if __name__ == "__main__":
    cli()
```

Wow! There are several new things happening in this code. First, you define the `-l` or `--long` option as a Boolean flag by setting its `is_flag` argument to `True`.

Inside `cli()`, you update the loop to produce a normal or long output depending on the user's choice. In the loop, you call the `build_output()` helper function to generate the appropriate output for each case.

The `build_output()` function 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 an entry. To generate the detailed output, you use tools like `Path.stat()` and a `datetime` object with a custom string format.

With all this new code in place, you can give your custom `ls` app a try. Go ahead and run the following commands:

Shell

```
(venv) $ python ls.py -l sample/
  2609 Jul 13 15:27:59 lorem.md
   428 Jul 12 15:28:38 realpython.md
    44 Jul 12 15:26:49 hello.txt

(venv) $ python ls.py -l sample/ sample_copy/
sample/:
  2609 Jul 12 15:27:59 lorem.md
   428 Jul 12 15:28:38 realpython.md
    44 Jul 12 15:26:49 hello.txt

sample_copy/:
  2609 Jul 12 15:27:18 lorem.md
   428 Jul 12 15:28:48 realpython.md
    44 Jul 12 15:27:18 hello.txt

(venv) $ python ls.py sample/ sample_copy/
sample/:
lorem.md     realpython.md     hello.txt

sample_copy/:
lorem.md     realpython.md     hello.txt
```

When you run the `ls.py` script with the `-l` flag, you get a detailed output of all the entries in the target directory. If you run it without the flag, then you get a short output.

Creating Feature Switches

In addition to Boolean flags, Click also supports what it calls feature switches. As the name suggests, this type of option allows you to enable or disable a given feature in your CLI apps. To define a feature switch, you'll have to create at least two options for the same parameter.

For example, consider the following update to your `upper_greet.py` app:

Python

```python
# upper_greet.py v2

import click

@click.command()
@click.argument("name", default="World")
@click.option("--upper", "casing", flag_value="upper")
@click.option("--lower", "casing", flag_value="lower")
def cli(name, casing):
    message = f"Hello, {name}!"
    if casing == "upper":
        message = message.upper()
    elif casing == "lower":
        message = message.lower()
    click.echo(message)

if __name__ == "__main__":
    cli()
```

The new version of `upper_greet.py` has two options: `--upper` and `--lower`. Both of these options operate on the same parameter, `"casing"`. You pass this parameter as an argument to the `cli()` function.

Inside `cli()`, you check the current value of `casing` and make the appropriate message transformation. If the user doesn't provide one of these options at the command line, then the app will display the message using its original casing:

Shell

```
(venv) $ python upper_greet.py --upper
HELLO, WORLD!

(venv) $ python upper_greet.py --lower
hello, world!

(venv) $ python upper_greet.py
Hello, World!
```

The `--upper` switch allows you to enable the uppercasing feature. Similarly, the `--lower` switch lets you use the lowercasing feature of your app. If you run the app with no switch, then you get the message with its original casing.

It's important to note that you can make one of the switches the default behavior of your app by setting its `default` argument to `True`. For example, if you want the `--upper` option to be the default behavior, then you can add `default=True` to the option's definition. In the above example, you didn't do this because printing the message using its original casing seems to be the appropriate and less surprising behavior.

Getting an Option's Value From Multiple Choices

Click has a parameter type called `Choice` that allows you to define an option with a list of allowed values to select from. You can instantiate `Choice` with a list of valid values for the option at hand. Click will take care of checking whether the input value that you provide at the command line is in the list of allowed values.

Here's a CLI app that defines a choice option called `--weekday`. This option will accept a string with the target weekday:

Python

```python
# days.py

import click

@click.command()
@click.option(
    "--weekday",
    type=click.Choice(
        [
            "Monday",
            "Tuesday",
            "Wednesday",
            "Thursday",
            "Friday",
            "Saturday",
            "Sunday",
        ]
    ),
)
def cli(weekday):
    click.echo(f"Weekday: {weekday}")

if __name__ == "__main__":
    cli()
```

In this example, you use the `Choice` class to provide the list of weekdays as strings. When your user runs this app, they'll have to provide a weekday that matches one of the values in the list. Otherwise, they'll get an error:

Shell

```
(venv) $ python days.py --weekday Monday
Weekday: Monday

(venv) $ python days.py --weekday Wednesday
Weekday: Wednesday

(venv) $ python days.py --weekday FRIDAY
Usage: days.py [OPTIONS]
Try 'days.py --help' for help.

Error: Invalid value for '--weekday': 'FRIDAY' is not one of 'Monday',
'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'.
```

The first two examples work as expected because the input values are in the list of allowed values. However, when you use `FRIDAY` in uppercase, you get an error because this value with that specific casing isn't in the list.

You have the possibility of working around this casing issue by setting the `case_sensitive` argument to `False` when you instantiate the `Choice` parameter type.

Getting Options From Environment Variables

Another exciting feature of Click options is that they can retrieve their values from environment variables. This feature can be pretty useful and may have several use cases.

For example, say that you're creating a CLI tool to consume a REST API. In this situation, you may need a secret key to access the API. One way to handle this key is by exporting it as an environment variable and making your app read it from there.

In the example below, you write a CLI app to retrieve cool space pictures and videos from NASA's main API page. To access this API, your application needs an API key that you can store in an environment variable and retrieve with Click automatically:

Python

```python
# nasa.py

import webbrowser

import click
import requests

BASE_URL = "https://api.nasa.gov/planetary"
TIMEOUT = 3

@click.command()
@click.option("--date", default="2021-10-01")
@click.option("--api-key", envvar="NASA_API_KEY")
def cli(date, api_key):
    endpoint = f"{BASE_URL}/apod"
    try:
        response = requests.get(
            endpoint,
            params={
                "api_key": api_key,
                "date": date,
            },
            timeout=TIMEOUT,
        )
    except requests.exceptions.RequestException as e:
        print(f"Error connecting to API: {e}")
        return

    try:
        url = response.json()["url"]
    except KeyError:
        print(f"No image available on {date}")
        return

    webbrowser.open(url)

if __name__ == "__main__":
    cli()
```

In this example, you import `webbrowser` from the Python standard library. This module allows you to quickly open URLs in your default browser. Then you import the `requests` library to make HTTP requests to the target REST API.

Note: To learn more about accessing REST APIs in your code, check out Python & APIs: A Winning Combo for Reading Public Data.

In the highlighted line, you create the `--api_key` option and set its `envvar` argument to `"NASA_API_KEY"`. This string represents the name that you'll use for the environment variable where you'll store the API key.

In the `cli()` function, you make an HTTP request to the `/apod` endpoint, get the target URL, and finally open that URL in your default browser.

Note: The `/apod` endpoint's name is an acronym that comes from **astronomy picture of the day**.

To give the above CLI app a try, go ahead and run the commands below. Note that in these commands, you'll use `"DEMO_KEY"` to access the API. This key has rate limits. So, if you want to create your own key, then you can do it on the API page:

Shell

```shell
(venv) $ export NASA_API_KEY="DEMO_KEY"
(venv) $ python nasa.py --date 2023-06-05
```

With the first command, you export the target environment variable. The second command runs the app. You'll see how your browser executes and shows an incredible image from space. Go ahead and play with different dates to retrieve some other amazing universe views.

It's important to note that you can also provide the key at the command line by explicitly using the `--api-key` option as usual. This comes in handy in situations where the environment variable is undefined.

Prompting the User for Input Values

Prompting the user for input is a pretty common requirement in CLI applications. Prompts can considerably improve your user's experience when they work with your app. Fortunately, Click has you covered with prompts as well.

With Click, you can create at least the following types of prompts:

- **Input** prompts
- **Password** prompts
- **Confirmation** prompts

You can create user prompts by using either the `prompt` argument to `@click.option` or the `click.prompt()` function. You'll also have dedicated decorators, such as `@click.password_option` and `@click.confirmation_option`, to create password and confirmation prompts.

For example, say that you need your application to get the user name and password at the command line to perform some restricted actions. In this case, you can take advantage of input and password prompts:

Python
```python
# user.py

import click

@click.command()
@click.option("--name", prompt="Username")
@click.option("--password", prompt="Password", hide_input=True)
def cli(name, password):
    if name != read_username() or password != read_password():
        click.echo("Invalid user credentials")
    else:
        click.echo(f"User {name} successfully logged in!")

def read_password():
    return "secret"

def read_username():
    return "admin"

if __name__ == "__main__":
    cli()
```

The `--name` option has a regular input prompt that you define with the `prompt` argument. The `--password` option has a prompt with the additional feature of hiding the input. This behavior is perfect for passwords. To set this new feature, you use the `hide_input` flag.

If you run this application from your command line, then you'll get the following behavior:

Shell
```shell
(venv) $ python user.py
Username: admin
Password:
User admin successfully logged in!
```

As you'll notice, the `Username` prompt shows the input value on the screen. In contrast, the `Password` prompt hides the input as you type, which is an appropriate behavior for password input.

When you're working with passwords, allowing the user to change their password may be a common requirement. In this scenario, you can use the `@click.password_option` decorator. This decorator allows you to create a password option that hides the input and asks for confirmation. If the two passwords don't match, then you get an error, and the password prompt shows again.

Here's a toy example of how to change the password of a given user:

Python

```python
# set_password.py

import click

@click.command()
@click.option("--name", prompt="Username")
@click.password_option("--set-password", prompt="Password")
def cli(name, set_password):
    # Change the password here...
    click.echo("Password successfully changed!")
    click.echo(f"Username: {name}")
    click.echo(f"Password: {set_password}")

if __name__ == "__main__":
    cli()
```

Using `@click.password_option`, you can create a password prompt that automatically hides the input and asks for confirmation. In this example, you create a `--set-password`, which does exactly that. Here's how it works in practice:

Shell

```shell
(venv) $ python set_password.py
Username: admin
Password:
Repeat for confirmation:
Error: The two entered values do not match.
Password:
Repeat for confirmation:
Password successfully changed!
Username: admin
Password: secret
```

In the first attempt to change the password, the initial input and the confirmation didn't match, so you got an error. The prompt shows again to allow you to enter the password again. Note that the prompt will appear until the two provided passwords match.

You can manually ask users for input. To do this, you can use the `prompt()` function. This function takes several arguments that allow you to create custom prompts and use them in other parts of your code, separate from where you defined the options.

For example, say that you want to create a command that adds two numbers together. In this case, you can have two custom prompts, one for each input number:

Python

```python
# add.py

import click

@click.command()
def cli():
    a = click.prompt("Enter an integer", type=click.INT, default=0)
    b = click.prompt("Enter another integer", type=click.INT, default=0)
    click.echo(f"{a} + {b} = {a + b}")

if __name__ == "__main__":
    cli()
```

In this example, you create two input prompts inside `cli()` using the `prompt()` function. The first prompt asks for the a value, while the second prompt asks for the b value. Both prompts will check that the input is a valid integer number and will show an error if not. If the user doesn't provide any input, then they can accept the default value, `0`, by pressing `Enter ↩`.

Here's how this app works:

Shell

```
(venv) $ python add.py
Enter an integer [0]: 42.0
Error: '42.0' is not a valid integer.
Enter an integer [0]: 42
Enter another integer [0]: 7
42 + 7 = 49
```

In the first input attempt, you enter a floating-point number. Click checks the input for you and displays an error message. Then you enter two valid integer values and get a successful result.

Click also provides a function called `confirm()`. This function comes in handy when you need to ask the user for confirmation to proceed with a sensitive action, such as deleting a file or removing a user.

The `confirm()` function prompts for confirmation with a *yes* or *no* question:

Python

```python
# remove.py

import click

@click.command()
@click.option("--remove-user")
def cli(remove_user):
    if click.confirm(f"Remove user '{remove_user}'?"):
        click.echo(f"User {remove_user} successfully removed!")
    else:
        click.echo("Aborted!")

if __name__ == "__main__":
    cli()
```

The `confirm()` function returns a Boolean value depending on the user's response to the yes or no confirmation question. If the user's answer is yes, then you run the intended action. Otherwise, you abort it.

Here's an example of using this app:

Shell

```
(venv) $ python remove.py --remove-user admin
Remove user 'admin'? [y/N]:
Aborted!

(venv) $ python remove.py --remove-user john
Remove user 'john'? [y/N]: y
User john successfully removed!
```

In the first example, you accept the default answer, `N` for *no*, by pressing `Enter ↩` as a reply to the prompt. Note that in CLI apps, you'll often find that the default option is capitalized as a way to indicate that it's the default. Click follows this common pattern in its prompts too.

In the second example, you explicitly respond yes by entering `y` and pressing `Enter ↩`. The app acts according to your response, either aborting or running the action.

Providing Parameter Types for Arguments and Options

In CLI development, arguments and options can take concrete input values at the command line. You've already learned that Click has some custom parameter types that you can use to define the type of input values. Using these parameter types, you can have type validation out of the box without writing a single line of code.

Here are some of the most relevant parameter types available in Click:

Parameter Type	Description
`click.STRING`	Represents Unicode strings and is the default parameter type for arguments and options

Parameter Type	Description
`click.INT`	Represents integer numbers
`click.FLOAT`	Represents floating-point numbers
`click.BOOL`	Represents Boolean values

Apart from these constants that represent primitive types, Click also has some handy classes that you can use to define other types of input values. You've already learned about the `click.Path`, `click.File`, `click.Choice`, and `click.Tuple` classes in previous sections. In addition to these classes, Click also includes the following:

Parameter Type Class	Description
`click.IntRange`	Restricts the input value to a range of integer numbers
`click.FloatRange`	Restricts the input value to a range of floating-point numbers
`click.DateTime`	Converts date strings into `datetime` objects

With all these custom types and classes, you can make your Click apps more robust and reliable. They'll also make you more productive because you won't have to implement type validation logic for your app's input values. Click does the hard work for you.

However, if you have specific validation needs, then you can create a custom parameter type with your own validation strategies.

Creating Subcommands in Click

Nested commands, or **subcommands**, are one of the most powerful and distinctive features of Click. What's a subcommand anyway? Many command-line applications, such as `pip`, pyenv, Poetry, and `git`, make extensive use of subcommands.

For example, `pip` has the `install` subcommand that allows you to install Python packages and libraries in a given environment. You used `pip install click` at the beginning of this tutorial to install the Click library, for example.

Similarly, the `git` application has many subcommands, such as `pull`, `push`, and `clone`. You'll likely find several examples of CLI applications with subcommands in your daily workflow because subcommands are pretty useful in real-world apps.

In the following section, you'll learn how to add subcommands to your Click applications using the `@click.group` decorator. You'll learn about two common approaches for creating subcommands:

1. **Registering subcommands right away**, which is appropriate when you have a minimal app in a single file
2. **Deferring subcommand registration**, which comes in handy when you have a complex app whose commands are distributed among multiple modules

Before you dive into the meat of this section, it's important to note that, in this tutorial, you'll only scratch the surface of Click's subcommands. However, you'll learn enough to get up and running with them in your CLI apps.

Registering Subcommand Right Away

To illustrate the first approach to creating subcommands in Click applications, say that you want to create a CLI app with four subcommands representing arithmetic operations:

1. **`add`** for adding two numbers together
2. **`sub`** for subtracting two numbers
3. **`mul`** for multiplying two numbers
4. **`div`** for dividing two numbers

To build this app, you start by creating a file called `calc.py` in your working directory. Then you create a command group using the `@click.group` decorator as in the code below:

Python

```python
# calc.py v1

import click

@click.group()
def cli():
    pass
```

In this piece of code, you create a command group called `cli` by decorating the `cli()` function with the `@click.group` decorator.

In this specific example, the `cli()` function provides the entry point for the app's CLI. It won't run any concrete operations. That's why it only contains a `pass` statement. However, other applications may need to take arguments and options in `cli()`, which you can implement as usual.

With the command group in place, you can start to add new subcommands right away. To do this, you use a decorator built using the group's name plus the `command()` function. For example, in the code below, you create the `add` command:

Python

```python
# calc.py v1

# ...

@cli.command()
@click.argument("a", type=click.FLOAT)
@click.argument("b", type=click.FLOAT)
def add(a, b):
    click.echo(a + b)

if __name__ == "__main__":
    cli()
```

In this code snippet, the decorator to create the `add` command is `@cli.command` rather than `@click.command`. This way, you're telling Click to attach the `add` command to the `cli` group.

At the end of the file, you place the usual name-main idiom to call the `cli()` function and start the CLI. That's it! Your `add` subcommand is ready for use. Go ahead and run the following command:

Shell

```shell
(venv) $ python calc.py add 3 8
11.0
```

Cool! Your `add` subcommand works as expected. It takes two numbers and adds them together, printing the result to your screen as a floating-point number.

As an exercise, you can implement the subcommands for the subtraction, multiplication, and division operations. Expand the collapsible section below to see the complete solution:

Subcommands Implementation Show/Hide

Once you've written the rest of the operations as subcommands, then go ahead and give them a try from your command line. Great, they work! But what if you don't have all your subcommands ready right off the bat? In that case, you can defer your subcommand registration, as you'll learn next.

Deferring Subcommand Registration

Instead of using the `@group_name.command()` decorator to add subcommands on top of a command group right away, you can use `group_name.add_command()` to add or register the subcommands later.

This approach is suitable for those situations where you have your commands spread into several modules in a complex application. It can also be useful when you need to build the CLI dynamically based on some configuration loaded from a file, for example.

Say that you refactor your `calc.py` application from the previous section, and now it has the following structure:

```
calc/
├── calc.py
└── commands.py
```

In this directory tree diagram, you have the `calc.py` and `commands.py` files. In the latter file, you've put all your commands, and it looks something like this:

Python

```python
# commands.py

import click

@click.command()
@click.argument("a", type=click.FLOAT)
@click.argument("b", type=click.FLOAT)
def add(a, b):
    click.echo(a + b)

@click.command()
@click.argument("a", type=click.FLOAT)
@click.argument("b", type=click.FLOAT)
def sub(a, b):
    click.echo(a - b)

@click.command()
@click.argument("a", type=click.FLOAT)
@click.argument("b", type=click.FLOAT)
def mul(a, b):
    click.echo(a * b)

@click.command()
@click.argument("a", type=click.FLOAT)
@click.argument("b", type=click.FLOAT)
def div(a, b):
    click.echo(a / b)
```

Note that in this file, you've defined the commands using the `@click.command` decorator. The rest of the code is the same code that you used in the previous section. Once you have this file with all your subcommands, then you can import them from `calc.py` and use the `.add_command()` method to register them:

Python

```python
# calc.py v2

import click
import commands

@click.group()
def cli():
    pass

cli.add_command(commands.add)
cli.add_command(commands.sub)
cli.add_command(commands.mul)
cli.add_command(commands.div)

if __name__ == "__main__":
    cli()
```

In the `calc.py` file, you first update the imports to include the `commands` module, which provides the implementations for your app's subcommands. Then you use the `.add_command()` method to register those subcommands in the `cli` group.

If you give this new version of your application a try, then you'll note that it works the same as its first version:

Shell

```
(venv) $ python calc/calc.py add 3 8
11.0
```

To run the app, you need to provide the new path because the app's entry-point script now lives in the `calc/` folder. As you can see, the functionality of `calc.py` remains the same. You've only changed the internal organization of your code.

In general, you'll use `.add_command()` to register subcommands when your CLI application is made of multiple modules and your commands are spread throughout those modules. For basic apps with limited functionalities and features, you can register the commands right away using the `@group_name.command` decorator, as you did in the previous section.

Tweaking Usage and Help Messages in a Click App

For CLI applications, it's crucial that you provide detailed documentation on how to use them. CLI apps don't have a graphical user interface for the user to interact with the app. They only have commands, arguments, and options, which are generally hard to memorize and learn. So, you have to carefully document all these commands, arguments, and options so that your users can use them.

Click has you covered in this aspect too. It provides convenient tools that allow you to fully document your apps, providing detailed and user-friendly help pages for them.

In the following sections, you'll learn how to fully document your CLI app using Click and some of its core features. To kick things off, you'll start by learning how to document commands and options.

Documenting Commands and Options

Click's commands and options accept a `help` argument that you can use to provide specific help messages for them. Those messages will show when you run the app with the `--help` option. To illustrate, get back to the most recent version of your `ls.py` script and check its current help page:

Shell

```
(venv) $ python ls.py --help
Usage: ls.py [OPTIONS] [PATHS]...

Options:
  -l, --long
  --help      Show this message and exit.
```

This help page is nice as a starting point. Its most valuable characteristic is that you didn't have to write a single line of code to build it. The Click library automatically generates it for you. However, you can tweak it further and make it more user-friendly and complete.

To start off, go ahead and update the code by adding a `help` argument to the `@click.command` decorator:

Python

```python
# ls.py v5

import datetime
from pathlib import Path

import click

@click.command(help="List the content of one or more directories.")
@click.option("-l", "--long", is_flag=True)
@click.argument(
    "paths",
    nargs=-1,
    type=click.Path(
        exists=True,
        file_okay=False,
        readable=True,
        path_type=Path,
    ),
)
def cli(paths, long):
    # ...
```

In the highlighted line, you pass a `help` argument containing a string that provides a general description of what the underlying command does. Now go ahead and run the app with the `--help` option again:

Shell

```
(venv) $ python ls.py --help
Usage: ls.py [OPTIONS] [PATHS]...

  List the content of one or more directories.

Options:
  -l, --long
  --help      Show this message and exit.
```

The app's help page looks different now. It includes a general description of what the application does.

It's important to note that when it comes to help pages for commands, the docstring of the underlying function will produce the same effect as the `help` argument. So, you can remove the `help` argument and provide a docstring for the `cli()` function to get an equivalent result. Go ahead and give it a try!

Note: If your app has multiple subcommands, then Click will add a `Commands` section to the help page and list each command. If you add help messages to all your subcommands, they'll appear beside the command's name.

As an exercise, you can add help messages to the subcommands of your `calc.py` app. Here's how the app's help page could look:

Shell

```
(venv) $ python calc.py --help
Usage: calc.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  add  Add two numbers.
  div  Divide two numbers.
  mul  Multiply two numbers.
  sub  Subtract two numbers.
```

On this help page, each subcommand shows its own help message, which is pretty helpful from the user's perspective.

Now update the `--long` option as in the code below to provide a descriptive help message:

Python

```python
# ls.py v6

import datetime
from pathlib import Path

import click

@click.command(help="List the content of one or more directories.")
@click.option(
    "-l",
    "--long",
    is_flag=True,
    help="Display the directory content in long format.",
)
@click.argument(
    "paths",
    nargs=-1,
    type=click.Path(
        exists=True,
        file_okay=False,
        readable=True,
        path_type=Path,
    ),
)
def cli(paths, long):
    # ...
```

In the definition of the `--long` option, you include the `help` argument with a description of what this specific option does. Here's how this change affects the app's help page:

Shell

```
(venv) $ python ls.py --help
Usage: ls.py [OPTIONS] [PATHS]...

  List the content of one or more directories.

Options:
  -l, --long  Display the directory content in long format.
  --help      Show this message and exit.
```

The `--long` option now includes a nice description that tells the user what its purpose is. That's great!

Documenting Arguments

Unlike `@click.command` and `@click.option`, the `click.argument()` decorator doesn't take a `help` argument. As the Click documentation says:

> This [the absence of a `help` argument] is to follow the general convention of Unix tools of using arguments for only the most necessary things, and to document them in the command help text by referring to them by name. (Source)

So, how can you document a command-line argument in a Click application? You'll use the docstring of the underlying function. Yes, it sounds weird. Commands also use that docstring. So, you'll have to refer to arguments by their names. For example, here's how you'd document the `PATHS` argument in your `ls.py` app:

Python

```python
# ls.py v6

import datetime
from pathlib import Path

import click

@click.command()
@click.option(
    "-l",
    "--long",
    is_flag=True,
    help="Display the directory content in long format.",
)
@click.argument(
    "paths",
    nargs=-1,
    type=click.Path(
        exists=True,
        file_okay=False,
        readable=True,
        path_type=Path,
    ),
)
def cli(paths, long):
    """List the content of one or more directories.

    PATHS is one or more directory paths whose content will be listed.
    """
    # ...
```

In this updated version of `ls.py`, you first remove the `help` argument from the command's definition. If you don't do this, then the docstring won't work as expected because the `help` argument will prevail. The docstring of `cli()` includes the original help message for the command. It also has an additional line that describes what the `PATHS` argument represents. Note how you've referred to the argument by its name.

Here's how the help page looks after these updates:

Shell

```shell
(venv) $ python ls.py --help
Usage: ls.py [OPTIONS] [PATHS]...

  List the content of one or more directories.

  PATHS is one or more directory paths whose content will be listed.

Options:
  -l, --long  Display the directory content in long format.
  --help      Show this message and exit.
```

This help page is looking good! You've documented the app's main command and the `PATHS` arguments using the docstring of the underlying `cli()` function. Now the app's help page provides enough guidance for the user to use it effectively.

Preparing a Click App for Installation and Use

When you dive into building CLI applications with Click, you quickly note that the official documentation recommends switching from the name-main idiom to `setuptools`. Using `setuptools` is the preferred way to install, develop, work with, and even distribute Click apps.

In this tutorial, you've used the idiom approach in all the examples to have a quick solution. Now it's time to do the switch into `setuptools`. In this section, you'll use the `calc.py` app as the sample project, and you'll start by creating a proper project layout for the app.

Create a Project Layout for Your CLI App

You'll use the `calc.py` script as the sample app to get into the `setuptools` switching. As a first step, you need to organize your code and lay out your CLI project. In the process, you should observe the following points:

- Create modules and packages to organize your code.
- Name the core package of a Python app after the app itself.
- Name each Python module according to its specific content or functionality.
- Add a `__main__.py` module to any Python package that's directly executable.

With these ideas in mind, you can use the following directory structure for laying out your `calc` project:

```
calc/
│
├── calc/
│   ├── __init__.py
│   ├── __main__.py
│   └── commands.py
│
├── pyproject.toml
└── README.md
```

The `calc/` folder is the project's root directory. In this directory, you have the following files:

- `pyproject.toml` is a TOML file that specifies the project's build system and many other configurations. In modern Python, this file is a sort of replacement for the `setup.py` script. So, you'll use `pyproject.toml` instead of `setup.py` in this example.
- `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 publish the project to PyPI as an open-source solution.

Then you have the `calc/` subdirectory, which holds the app's core package. Here's a description of its content:

- **`__init__.py`** enables `calc/` as a Python package. In this example, this file will be empty.
- **`__main__.py`** provides the application's entry-point script or executable file.
- **`commands.py`** provides the application's subcommands.

In the following collapsible sections, you'll find the content of the `__main__.py` and `commands.py` files:

`__main__.py` Show/Hide

`commands.py` Show/Hide

With the project layout in place, you're now ready to write a suitable `pyproject.toml` file and get your project ready for development, use, and even distribution.

Write a `pyproject.toml` File for Your Click Project

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 `calc` project:

TOML

```toml
# pyproject.toml

[build-system]
requires = ["setuptools>=65.5.0", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = "calc"
version = "0.1.0"
description = "A CLI application that performs arithmetic operations."
readme = "README.md"
authors = [{ name = "Real Python", email = "info@realpython.com" }]
dependencies = [
    "click >= 8.1.3",
]

[project.scripts]
calc = "calc.__main__:cli"
```

The `[build-system]` table header defines `setuptools` as your app's build system and specifies the dependencies for building the application.

The `[project]` header allows you to provide general metadata for the application. This metadata may include many key-value pairs, including the app's name, version, general description, and so on.

The `dependencies` key is quite important and convenient. Through this key, you can list all the project's dependencies and their target versions. In this example, the only dependency is Click, and you're using a version greater than or equal to `8.1.3`. The project's build system will take that list and automatically install all of its items.

Finally, in the `[project.scripts]` heading, you define the application's entry-point script, which is the `cli()` function in the `__main__.py` module in this example. With this final setup in place, you're ready to give the app a try. To do this, you should first create a dedicated virtual environment for your `calc` project.

Create a Virtual Environment and Install Your Click App

You already learned how to create a Python virtual environment. So, go ahead and open a terminal window. Then navigate to your `calc` project's root folder. Once you're in there, run the following commands to create a fresh environment:

Windows Linux + macOS

Windows Command Prompt

```
PS> python -m venv venv
PS> venv\Scripts\activate
```

Great! You have a fresh virtual environment within your project's folder. To install the application in there, you'll use the `-e` option of `pip install`. This option allows you to install packages, libraries, and tools in editable mode.

Editable mode lets you work on the source code while being able to try the latest modifications as you implement them. This mode is quite useful in the development stage.

Here's the command that you need to run to install `calc`:

Shell

```
(venv) $ python -m pip install -e .
```

Once you've run this command, then your `calc` app will be installed in your current Python environment. To check this out, go ahead and run the following command:

Shell

```
(venv) $ pip list
Package            Version
------------------ -------
calc               0.1.0
click              8.1.6
...
```

The `pip list` command lists all the currently installed packages in a given environment. As you can see in the above output, `calc` is installed. Another interesting point is that the project dependency, `click`, is also installed. Yes, the `dependencies` key in your project's `pyproject.toml` file did the magic.

From within the project's dedicated virtual environment, you'll be able to run the app directly as a regular command:

Shell

```
(venv) $ calc add 3 4
7.0

(venv) $ calc sub 3 4
-1.0

(venv) $ calc mul 3 4
12.0

(venv) $ calc div 3 4
0.75

(venv) $ calc --help
Usage: calc [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  add  Add two numbers.
  div  Divide two numbers.
  mul  Multiply two numbers.
  sub  Subtract two numbers.
```

Your `calc` application works as a regular command now. There's a detail that you should note on the app's help page at the end of the examples above. The `Usage` line now shows the app's name, `calc`, instead of the Python filename, `calc.py`. That's great!

You can try to extend the app's functionalities and maybe add more complex math operations as an exercise. Go ahead and give it a try!

Conclusion

Now you have a broad understanding of how the **Click** library works in Python. You know how to use it to create powerful command-line interfaces for small or large applications and automation tools. With Click, you can quickly create apps that provide arguments, options, and subcommands.

You've also learned how to tweak your app's help page, which can fundamentally improve your user experience.

In this tutorial, you've learned how to:

- Build **command-line interfaces** for your apps using **Click** and Python
- Support **arguments**, **options**, and **subcommands** in your CLI apps
- Enhance the **usage** and **help pages** of your CLI apps
- Prepare a Click app for **installation**, **development**, **use**, and **distribution**

The Click library provides a robust and mature solution for creating extensible and powerful CLIs. Knowing how to use this library will allow you to write effective and intuitive command-line applications, which is a great skill to have as a Python developer.

Get Your Code: Click here to download the sample code that you'll use to build your CLI app with Click and Python.

Mark as Completed

About **Leodanis Pozo Ramos**

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

» More about Leodanis

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:

Aldren

Bartosz

Geir Arne

Kate

What Do You Think?

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: **devops** **intermediate** **python** **tools**