How to Write Python Command-Line Interfaces like a Pro

Simon Hawe
Nov 8 · 8 min read ★

Photo by Kelly Sikkema on Unsplash

We as Data Scientists face doing many repetitive and similar tasks. That includes creating weekly reports, executing extract, transform, load (ETL) jobs, or training

models using different parameter sets. Often, we end up having a bunch of Python scripts, where we change parameters in code every time we run them. I hate doing this! That's why I got into the habit of transforming my scripts into reusable command-line interface (CLI) tools. This increased my efficiency and made me more productive in my daily life. I started doing this using Argparse but this was not enjoyable as I had to produce a lot of ugly code. So I thought, can't I achieve that without having to write a lot of code over and over again? Can I even enjoy writing CLI tools?

> *Click is your friend!*

So what is Click? From the webpage:

> *It (Click) aims to make the process of writing command-line tools quick and fun while also preventing any frustration caused by the inability to implement an intended CLI API.*

To me, that sounds great, doesn't it?

In this article, I give you a hands-on guide on how to build Python CLIs using Click. I build up an example step by step that shows you the basic features and benefits Click offers. After this tutorial, you should be able to write your next CLI tool with joy and in a blink of an eye :) So let's get our hands dirty!

The Tutorial

In this tutorial, we build up a Python CLI using Click that evolves step by step. I start with the basics, and with each step, I introduce a new concept offered by Click. Apart from Click, I use Poetry to manage dependencies and packages.

Preparation

First, let's install Poetry. There are various ways of doing that, see my article, but here we use pip

```
pip install poetry==0.12.7
```

Next, we use Poetry to create a project named *cli-tutorial,* add click and funcy as a dependency, and create a file cli.py that we later fill with code

```
poetry new cli-tutorial
cd cli-tutorial
poetry add click funcy
# Create the file we will put in all our code
touch cli_tutorial/cli.py
```

I have added funcy, as I will make use of it later. To see what that module is good for, I refer the interested reader to this article. Now, we are ready to go and implement our first CLI. As a side note, all example code is available on my GitHub account.

Our First Click CLI

Our initial CLI reads a CSV file from disk, processes it (how is not important for this tutorial), and stores the result in an Excel file. Both, the path to the input-, and the output file should be configurable by the user. The user *must* specify the input file path. Specifying the output file path is optional and defaults to *output.xlsx*. Using Click, the code that does that reads as

```
import click

@click.command()
@click.option("--in", "-i", "in_file", required=True,
    help="Path to csv file to be processed.",
)
@click.option("--out-file", "-o", default="./output.xlsx",
    help="Path to excel file to store the result.")
def process(in_file, out_file):
    """ Processes the input file IN and stores the result to
    output file OUT.
    """
    input = read_csv(in_file)
    output = process_csv(input)
    write_excel(output, out_file)

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

What do we do here?

1. We decorate the method *process* that we want to invoke from the command line with *click.command*.

2. We define the command-line arguments using the *click.option* decorator. Now, you must be careful using the correct argument names in your decorated function. If we

add a string without dashes to *click.option*, the argument must match that string. This is the case for --in and in_file. If all names contain leading dashes, Click generates the argument name using the longest name and converting all non-leading dashes to underscores. The name is converted to lowercase. This is the case for --out-file and out_file. For more details, consult the Click documentation.

3. We configure desired prerequisites like defaults or required arguments using the corresponding arguments to *click.option*.

4. We add a help text to our arguments, which is shown when invoking our function using --help. The docstring from our function will also be shown there.

Now, you can invoke this CLI in various ways

```
# Prints help
python -m cli_tutorial.cli --help
# Use single char -i for loading the file
python -m cli_tutorial.cli -i path/to/some/file.csv
# Specify both file with long name
python -m cli_tutorial.cli --in path/to/file.csv --out-file out.xlsx
```

Cool, we have created our first CLI using Click!

Note that, I have not implemented *read_csv*, *process_csv*, and *write_excel* but assume they exist and do what they are supposed to do.

One issue with CLIs is that we pass parameters as generic strings. Why is that an issue? Because these strings must be parsed to the actual types, which can fail due to badly

formatted user input. Look at our example where we use paths and try to load a CSV file. A user can provide a string that does not represent a path at all. And even if the string is formatted correctly, the corresponding file might not exist or you don't have the right permission to access it. Wouldn't it be desirable to automatically validate the input, parse it if possible or fail early with helpful error messages? Ideally, all that without having to write a lot of code? Click supports us with this by specifying types for our arguments.

Type Specification

In our example CLI, we want the user to pass in a **valid path** to an **existing file** for which we have **read permissions**. If these requirements are fulfilled, we can load the input file. Additionally, if the user specifies an output file path, this should be a valid path. We can enforce all that by passing a *click.Path* object to the *type* argument of the *click.option* decorator

```
@click.command()
@click.option("--in", "-i", "in_file", required=True,
    help="Path to csv file to be processed.",
    type=click.Path(exists=True, dir_okay=False, readable=True),
)
@click.option("--out-file", "-o", default="./output.csv",
    help="Path to csv file to store the result.",
    type=click.Path(dir_okay=False),
)
def process(in_file, out_file):
    """ Processes the input file IN and stores the result to output
    file OUT.
    """
    input = read_csv(in_file)
    output = process_csv(input)
    write_excel(output, out_file)
...
```

click.Path is one of the various types offered by Click out of the box. You can also implement custom types, but this is out of scope for this tutorial. For more details, I refer the interested readers to the Click documentation.

Boolean Flags

Another helpful feature offered by Click is boolean flags. Probably, the most famous boolean flag is the *verbose* flag. If set to true, your tool will print out a lot of information to the terminal. If set to false, only a few things are printed. With Click, we can implement that as

```
from funcy import identity
...
@click.option('--verbose', is_flag=True, help="Verbose output")
def process(in_file, out_file, verbose):
    """ Processes the input file IN and stores the result to
    output file OUT.
    """
    print_func = print if verbose else identity
    print_func("We will start with the input")
    input = read_csv(in_file)
    print_func("Next we procees the data")
    output = process_csv(input)
    print_func("Finally, we dump it")
    write_excel(output, out_file)
```

All you have to do is add another *click.option* decorate and set *is_flag=True*. Now, to get a verbose output, you need to call the CLI as

```
python -m cli_tutorial.cli -i path/to/some/file.cs --verbose
```

Feature Switch

Assume we do not only want to store the result of *process_csv* locally, but we also want to upload it to a server. Additionally, there is not only one target server but a development-, a testing-, and a production instance. You can access these three instances via different URLs. One option for a user to select the server is to pass the full URL as an argument, which she has to type in. But, this is not only error-prone but also a tedious job. In situations like that, I use *feature switches* to simplify the user's life. What they do is best explained through code

```
...
@click.option(
    "--dev", "server_url", help="Upload to dev server",
    flag_value='https://dev.server.org/api/v2/upload',
)
@click.option(
    "--test", "server_url", help="Upload to test server",
    flag_value='https://test.server.com/api/v2/upload',
)
@click.option(
    "--prod", "server_url", help="Upload to prod server",
    flag_value='https://real.server.com/api/v2/upload',
    default=True
)
def process(in_file, out_file, verbose, server_url):
```

```
    """ Processes the input file IN and stores the result to output
    file OUT.
    """
    print_func = print if verbose else identity
    print_func("We will start with the input")
    input = read_csv(in_file)
    print_func("Next we procees the data")
    output = process_csv(input)
    print_func("Finally, we dump it")
    write_excel(output, out_file)
    print_func("Upload it to the server")
    upload_to(server_url, output)
...
```

Here, I have added three *click.option* decorators for the three possible server URLs. The important bit is, that all three options have the same target variable *server_url*. Depending on which option you choose, the value of *server_url* is equal to the corresponding value defined in *flag_value*. You chose one of those by adding *--dev*, *--test*, or *--prod* as an argument. So when you execute

```
python –m cli_tutorial.cli –i path/to/some/file.csv ––test
```

server_url is equal to 'https://test.server.com/api/v2/upload'. If we don't specify any of the three flags, Click takes the value of *--dev*, as I set *default=True*.

Username & Password Prompts

Unfortunately, or rather luckily :), our servers are password protected. So to upload our file, we need a username and a password. For sure, you can provide those as a standard click.option arguments. However, your password ends up in your command history in plain text. This can become a security issue.

We like a prompt for the user to type his password without echoing it to the terminal and without storing it in the command history. For the username, we also want a simple prompt *with* echoing. Nothing easier than that when you know Click. Here is the code.

```
import os
...
@click.option('––user', prompt=True,
              default=lambda: os.environ.get('USER', ''))
@click.password_option()
def process(in_file, out_file, verbose, server_url, user, password):
```

```
    ...
    upload_to(server_url, output, user, password)
```

To add a prompt for an argument, you have to *set prompt=True*. This will *add* a prompt whenever a user does not specify the *--user* argument, but she can still specify it like that. The *default* value is used, when you just hit enter on the prompt. The default is determined using a function, which is another handy feature offered by Click.

Prompting passwords without echoing it to the terminal and asking for a confirmation is so common, that Click offers a dedicated decorator called *password_option*. An important note; this will not prevent the user to pass the password via --password MYSECRETPASSWORD. It just enables her not doing that.

And that's it. We've build the full CLI. Before we call it a day, I like to give you a final hint in the next section.

Poetry Scripts

A final tip that I want to give you, that is not related to Click but perfectly matches the CLI topic, is creating Poetry scripts. With Poetry scripts, you can create executables to invoke your Python functions from the command line, as you can do with Setuptools scripts. So how does that look like? First, you need to add the following to your pyproject.toml file

```
[tool.poetry.scripts]
your-wanted-name = 'cli_tutorial.cli:process'
```

The *your-wanted-name* is an alias for the function *process* defined in the module *cli_tutorial.cli*. Now, you can invoke that through

```
poetry run your-wanted-name -i ./dummy.csv --verbose --dev
```

This allows you, for example, to add multiple CLI functions to the same file, to define aliases, and you don't have to add an *if __name__ == "__main__"* block.

Wrap Up

In this article, I showed you how to use Click and Poetry to build CLI tools with joy and become more productive. This was just a small subset of the features offered by Click. There are many others like callbacks, nested commands, or choice options just to mention a few. For now, I refer the interested reader to the Click documentation but I might write a follow-up post covering these advanced topics. Stay tuned and thank you for following along this post. Feel free to contact me for questions, comments, or suggestions.

Python Data Science Tools Productivity Programming