A Simple Guide To Command Line Arguments With ArgParse

Your go-to guide to get argparse quickly up and running

Sam Starkman Dec 4, 2020 · 7 min read ⋆

Photo by Dan Gold on Unsplash

The standard Python library `argparse` used to incorporate the parsing of command line arguments. Instead of having to manually set variables inside of the code, `argparse` can be used to add flexibility and reusability to your code by allowing user input values to be parsed and utilized.

Installation

Since `argparse` is part of the standard Python library, it should already be installed. However, if it's not, you can install it using the follwing command:

```
pip install argparse
```

If you do not have `pip` installed, follow the installation docs here.

Getting Started

Here is a file called `hello.py` to demonstrate a very basic example of the structure and usage of the `argparse` library:

```
# Import the library
import argparse

# Create the parser
parser = argparse.ArgumentParser()

# Add an argument
parser.add_argument('--name', type=str, required=True)

# Parse the argument
args = parser.parse_args()

# Print "Hello" + the user input argument
print('Hello,', args.name)
```

The code above is the most straightforward way to implement `argparse`. After importing the library, `argparse.ArgumentParser()` initializes the parser so that you can start to add custom arguments. To add your arguments, use `parser.add_argument()`. Some important parameters to note for this method are `name`, `type`, and `required`. The `name` is exactly what it sounds like — the name of the command line field. The `type` is the variable type that is expected as an input, and the `required` parameter is a boolean for whether or not this command line field is mandatory or not. The actual arguments can be accessed with `args.name`, where `name` is the name of the argument identified in `add_argument()`.

Below is an example of the output when this code is run:

```
C:/> python hello.py

usage: hello.py [-h] --name NAME
hello.py: error: the following arguments are required: --name
```

As you can see, this will throw an error because the required `name` argument is missing. This is the result when you include `--name`:

```
C:/> python hello.py --name Sam
```

```
Hello, Sam
```

Positional Arguments

Sometimes, you don't want to use the flag's name in the argument. You can use a positional argument to eliminate the need to specify the `--name` flag before inputting the actual value. Below are two versions — the first without positional arguments (`multiply.py`), and the second using positional arguments (`multiply_with_positional.py`).

multiply.py — Without Positional Arguments

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--x', type=int, required=True)
parser.add_argument('--y', type=int, required=True)
args = parser.parse_args()

product = args.x * args.y
print('Product:', product)
```

Output

```
C:/> python multiply.py --x 4 --y 5

Product: 20
```

multiply_with_positional.py — With Positional Arguments

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('x', type=int)
parser.add_argument('y', type=int)
args = parser.parse_args()

product = args.x * args.y
print('Product:', product)
```

Output

```
C:/> python multiply_with_positional.py 4 5

Product: 20
```

While positional arguments make the command line cleaner, it can sometimes be difficult to tell what the actual field is since there is no visible name associated with it. To aid with this, you can use the `help` parameter in `add_argument()` to specify more details about the argument.

```
parser.add_argument('x', type=int, help='The first value to
multiply')

parser.add_argument('y', type=int, help='The second value to
multiply')
```

Now, when you run your program with the help flag `-h`, you can see these details.

```
C:/> python multiply.py -h

usage: multiply.py [-h] x y

positional arguments:
x           The first value to multiply
y           The second value to multiply
```

Optional Arguments

Optional arguments are useful if you want to give the user a choice to enable certain features. To add an optional argument, simply omit the `required` parameter in `add_argument()`.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--name', type=str, required=True)
parser.add_argument('--age', type=int)
args = parser.parse_args()

if args.age:
  print(args.name, 'is', args.age, 'years old.')

else:
  print('Hello,', args.name + '!')
```

Here we see two arguments: a name, which is required, and an optional age. (Notice the `required=True` is missing from the `--age` argument.)

The two outputs below show the execution when `--age` is included and when it is not.

With Optional Argument

```
C:/> python optional.py --name Sam --age 23

Sam is 23 years old.
```

Without Optional Argument

```
C:/> python optional.py --name Sam

Hello, Sam!
```

We can check to see if the `args.age` argument exists and implement different logic based on whether or not the value was included.

Multiple Input Arguments

Let's say that instead of specifying `x` and `y` arguments for the user to input, you want the user to input a list of numbers and the script will return the sum of them all. There is no need to create a new argument for each new value (you also can't possibly know how many values a user will input!) Using the `nargs` parameter in `add_argument()`, you can specify the number (or arbitrary number) of inputs the argument should expect.

In this example named `sum.py`, the `--value` argument takes in 3 integers and will print the sum.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--values', type=int, nargs=3)
args = parser.parse_args()

sum = sum(args.values)
print('Sum:', sum)
```

Output

```
C:/> python sum.py --values 1 2 3

Sum: 6
```

What if you don't want just 3 values, but any number of inputs? You can set `nargs='+'`, which will allow the argument to take in any number of values. Using the same example above, with the only change being `nargs=3` to `nargs='+'`, you can run the script with however many input values you want.

```
C:/> python sum.py --values 1 2 3

Sum: 6

C:/> python sum.py --values 2 4 6 8 10

Sum: 30

C:/> python sum.py --values 1 2 3 4 5 6 7 8 9 10

Sum: 55
```

Mutually Exclusive Arguments

Another important `argparse` feature is mutally exclusive arguments. There are times that, depending on one argument, you want to restrict the use of another. This could be because the user should only need to use one of the arguments, or that the arguments conflict with each other. The method `add_mutually_exclusive_group()` let's us do exactly that — add a group of arguments that are mutually exclusive.

This next example, `mutually_exclusive.py`, demonstrates how both arguments in a mutually exclusive group cannot be used at the same time.

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument('--add', action='store_true')
group.add_argument('--subtract', action='store_true')
parser.add_argument('x', type=int)
parser.add_argument('y', type=int)
args = parser.parse_args()

if args.add:
  sum = args.x + args.y
  print('Sum:', sum)

elif args.subtract:
  difference = args.x - args.y
  print('Difference:', difference)
```

By creating a group with `group = parser.add_mutually_exclusive_group()`, the user is only allowed to select one of the arguments to use. In the `add_argument()` method, there is a new parameter called `action`. This is simply storing the default method if the argument is blank.

The output example below shows what happens when you try to call `--add` and `--subtract` in the same command.

Output

```
C:/> python mutually_exclusive.py --add 1 2

Sum: 3

C:/> python mutually_exclusive.py --subtract 4 3

Difference: 1

C:/> python mutually_exclusive.py --add --subtract 4 3

usage: mutually_exclusive.py [-h] [--add | --subtract] x y
mutually_exclusive.py: error: argument --subtract: not allowed with
argument --add
```

As you can see from the error message, the script does not allow for `--add` and `--subtract` to be called at the same time!

Subparsers

The last `argparse` feature I am going to discuss is subparsers. Subparsers are powerful in that they allow for different arguments to be permitted based on the command being run. For example, when using the `git` command, some options are `git checkout`, `git commit`, and `git add`. Each one of these commands requires a unique set of arguments, and subparsers allow you to distinguish between them.

This last example describes how to create a subparser to establish completely different sets of arguments, depending on the command run. This `user.py` script will be used to demonstrate a login or a register, contingent on the first positional argument provided.

```
import argparse

parser = argparse.ArgumentParser()
subparser = parser.add_subparsers(dest='command')
login = subparser.add_parser('login')
register = subparser.add_parser('register')
```

```
login.add_argument('--username', type=str, required=True)
login.add_argument('--password', type=str, required=True)

register.add_argument('--firstname', type=str, required=True)
register.add_argument('--lastname', type=str, required=True)
register.add_argument('--username', type=str, required=True)
register.add_argument('--email', type=str, required=True)
register.add_argument('--password', type=str, required=True)

args = parser.parse_args()

if args.command == 'login':
  print('Logging in with username:', args.username,
  'and password:', args.password)

elif args.command == 'register':
  print('Creating username', args.username,
  'for new member', args.firstname, args.lastname,
  'with email:', args.email,
  'and password:', args.password)
```

Here, we added `subparser = parser.add_subparsers(dest='command')`. This is used to create the subparser, and the `dest='command'` is used to differentiate between which argument is actually used. You can see in the `if` statement that we distinguish between "login" and "register" with `args.command`.

We create two separate subparsers — one for "login" and one for "register". This allows us to add individual arguments to each. Here, the "login" subparser requires a username and a password. The "register" subparser takes in a first and last name, a username, an email, and a password. Depending on whether "login" or "register" is specified in the script, the user must input the correct arguments, and the conditional statement will confirm the results.

Login

```
C:/> python user.py login --username D0loresh4ze --password
whoismrrobot

Logging in with username: D0loresh4ze and password: whoismrrobot
```

Register

```
C:/> python user.py register --firstname Dolores --lastname Haze --
username Doloresh4ze --email dhaze@ecorp.com --password whoismrrobot

Creating username Doloresh4ze for new member Dolores Haze with email:
dhaze@ecorp.com and password: whoismrrobot
```

As expected, the print statement for logging in is called when `login` is specified as the command argument, and the register print statement is called when `register` is specified with its respective arguments.

Takeaways

The goal of this post was to give a brief and relevant overview of the Python library `argparse`. Code cleanliness and reusability is extremely valuable these days, and `argparse` can help developers write more modular scripts when requiring user input and interaction.

Thank you all for reading — I hope that I was able to improve your understanding of `argparse`, and that you can utilize it in your next project!

Thanks to Linda Chen.

Python Data Science Programming Tutorial Software Development