Script Tip: `argparse` (How to make your own CLI)

A huge part of any Python developer's arsenal is the ability to link your Python program with "other stuff". "other stuff" meaning everything from other programs to other humans to other developers. As your careers develop you learn how to write more something-Is. The most common ones will be:

- APIs - "Application Programming Interface"s (let programs talk to each other)
- GUIs - "Graphical User Interface"s (let humans talk to the program)
- CLIs - "Command Line Interface"s (let humans talk to the program but its easy to develop)

Of the three, CLIs are by far the easiest to build conceptually and programmatically. Crafting good GUIs and APIs is very hard and is a fine art that requires many years of training. As you'll find going through life, most people aren't very good at it.

CLIs on the other hand are *fine*; it's hard to deliberately mess one up, so long as it's not too complicated. And that's where this script tip comes in! In this article I'll explain how the builtin Python CLI-builder `argparse` works. The perfect tool for building a quick and easy interface with your Python script.

Introducing `argparse`

`argparse` is the builtin library to create CLIs with in Python. It is not the best, nor the simplest, but it is *always* there, and as such it is useful to know about this one because you may occasionally find yourself unable to install external packages for some reason and be forced to use it (this happens a lot commercially in my experience; usually for licensing reasons).

If you are open to 3rd parties then I suggest checking out the following options:

- `click`
- `docopt`
- `invoke`

If this is a bit overwhelming Real Python has written a very nice article comparing all three AND argparse, which you should check out here.

This article, however, will focus on `argparse` and is aimed at people coming across it for the first time.

What does a CLI "look like"?

A CLI has the following components: "the command" (`python` in Python, or some other alias like `py2` or `py3` are common), "the option(s)" (flags that indicate what you want to do. The most common being the `--help` flag.), and "the argument(s)" (information you want to insert into the script, like the path to a file, for example). In vanilla Python there is one "command", the call to `python` and at least one argument every time, the script name/path.

```
python my_script.py
```

The command here is `python` and the argument is `my_script.py`.

`sys.argv`

OK but that's all on the command line. How does Python access this information inside a script? Well, in Python, the builtin `sys` package has the property `argv`, which is a list of all the input arguments as strings. The delimiter separating the arguments is the space character, although substrings within the CLI command are preserved (strings contained within a pair of double quotes `"..."`).

If you have a script (`script.py`) containing

```
import sys

print(sys.argv)
```

and run it, you get the following output.

```
>>> python script.py
['.\\script.py']
```

> Note: the double backslash is because I'm on Windows, which uses backslashes in paths, and backslash is also the escape character in Python strings, so it has to be escaped to be used in a path. As such, path visualisations in Python on Windows will be positively FULL of `\\`. Try to use pathlib where you can to get around this somewhat!

If we add more arguments to our command line we can see just how `argv` works.

```
>>> python script.py new set-of "test args"
['script.py', 'new', 'set-of', 'test args']
```

> Why is the backslash gone? Honestly, I'm not sure and I couldn't find any info after a cursory google. If you have any idea why, please leave a comment! Although the reason is not important for this article because for the CLI the only argument we don't care about is the first one because it is always the same!

So, `argparse` is essentially a package that parses `sys.argv` and makes sense of it, turning it into options and arguments for the CLI.

> Note: All input on the command line are *command line arguments* but in the CLI, only some of those are *CLI arguments*. Some are what I'm calling *options* or *flags*.

Nitty-gritty of CLI

Generally CLIs make use of option-argument pairs. You provide some sort of option or flag and then the input corresponding to it, like `--density 3.4` or `--path .\\Program\\Python\\`, and this makes sense. For example

```
>>> python script2.py --factorial 9
362880
```

the option is `--factorial` and its corresponding argument is `9`. Providing additional args throws an error, unless we were to modify the programn to handle it.

Scripts are not just limited to single pair either. We can write one to have multiple pairs as well:

```
>>> python .\script3.py --base 2 --exponent 3
8.0
```

In fact, you can have multiple pairs that are *optional*.

```
>>> python .\script4.py --num1 3. --num2 4.
1.0
1.3333333333333333
>>> python .\script4.py --num1 5.
1.6666666666666667
```

> Note: the `--` is a convention. Generally `--` is used before full words and a single `-` is used for the same argument but with a single character alias. This is most easily demonstrated with the example that `--help` and `-h` are the same.
>
> ```
> >>> python .\script4.py -h
> usage: script4.py [-h] [--num1 NUM1] [--num2 NUM2]
>
> Divide by 3
>
> options:
>   -h, --help   show this help message and exit
>   --num1 NUM1  Option 1
>   --num2 NUM2  Option 2
>
> >>> python .\script4.py --help
> usage: script4.py [-h] [--num1 NUM1] [--num2 NUM2]
>
> Divide by 3
>
> options:
>   -h, --help   show this help message and exit
>   --num1 NUM1  Option 1
>   --num2 NUM2  Option 2
> ```

Constructing a CLI with `argparse`

OK so we're ready to tackle the main topic of this tip now; apologies if you already knew all of the above. I like to cover all my bases in an article because one's work may well be someone's first exposure to any number of concepts, so it always pays to be thorough.

Anyway! How do we use argparse? It's pretty simple, but requires a bit of mental gymnastics if you aren't used to thinking about programming this way.

Your program executes in the following way:

1. You smash that Enter button on the command line
2. The arguments you added are all stored in `sys.argv`
3. The program enters your script and starts executing things

You need to build the parser into the program so your program has actually already parsed the args, but the parser hasn't been executed? Argh!

It's OK. The program actually stores your args in `sys.argv` as I said before, and then your parser parses *that*.

So your program's execution order should be structured like so

1. Perform your imports (imports should always be at the top of the file in Python)
2. Construct the parser
3. Parse the args
4. Perform logic based on the parsed args
5. Other stuff

How do I construct the parser?

First of all you create an instance of `argparse.ArgumentParser`.

```
parser = ArgumentParser(description='Demo program')
```

The description is purely for documentation purposes and you can leave it out if you want, though why would you?! Writing docs is fun!

Next we add arguments to our parser. We tell the parser what it should expect to see on the command line and how to treat those inputs. For this tutorial though we're just keeping it as simple as possible and not worrying about the many things you can do.

```
parser.add_argument('--density', help='density provided as a float')
```

this adds the "option" `--density` which is a point of entry for the "argument" associated with it.

Now we parse the args, which stores all the supplied arguments in the returned namespace object that I always call `args`.

```
args = parser.parse_args()
```

> Note: If no arguments are provided to `parse_args` then it will use `sys.argv`, but if you *do* provide a string it will actually parse that instead! It's a neat little trick you can make use of when building argparse in non-standard locations like jupyter notebooks.

Now we have `args` we can get at all the inputs! For now let's just print the density.

```
print(args.density)
```

> Note: `argparse` does some magic behind the scenes and strips away the `--`, so we don't have to worry about that.

OK so what happens if we run our program?

```
>>> python script5.py --density 4.0
4.0
```

And what if we run it without arguments?

```
>>> python script5.py
None
```

This happens because if we don't provide `density` the namespace with it is still created. In fact, we can use `argparse` to specify the default value should nothing be supplied. Which we would do by changing the `add_argument` line to

```
parser.add_argument('--density', default=0., help='density provided as a float')
```

which specifies the default as `0.0` if nothing else is supplied.

And that's pretty much it!

Some tips 'n' tricks about `argparse`

- You do not need to add an argument for `--help`. This comes with `argparse`, and will always be present when creating a CLI with it.
- When you add an argument in `argparse` you can specify the `type` that argument should be converted to, like `float` or `int`. If no type is provided the output type will always be a string unless it was not provided, which case the result will be `None` of the `NoneType`.
- By specifying the `action` of an argument as `"store_true"` you can create an option that is parsed as `True` if it is present and `False` if it is not. If the `action` is set to `"store_false"` instead the reverse is

achieved!

```
parser.add_argument('--flag', action='store_true', help='if provided, set to True,
otherwise False')
```

```
>>> python script6.py --flag --reverse-flag
True
False
>>> python script6.py
False
True
```

- You can add an easy versioning return with the `"version"` action, and the version to be printed can be specified with the `version` kwarg. Like using the builtin `--help` option, the program will print the version and auto-quit afterward.

```
parser.add_argument('--version', action='version', version='script6 v1.0')
```

```
>>> python script6.py --version
script6 v1.0
```

- When developing programs with `argparse` it is best practice to group your code into functions, often in a separate file. In `script7.py` I've grouped things as they commonly are.

```
from argparse import ArgumentParser


def create_parser():
    parser = ArgumentParser(description='Demo program')
    parser.add_argument('--flag',
                        action='store_true',
                        help='when provided sets to True, otherwise False')
    parser.add_argument('--reverse-flag',
                        action='store_false',
                        help='when provided sets to False, otherwise True')

    parser.add_argument('--version', action='version', version='script6 v1.0')
    return parser


def cli():
    p = create_parser()
    args = p.parse_args()
```

```
    return args.flag, args.reverse_flag


def main():
    flag, reverse_flag = cli()
    print(flag)
    print(reverse_flag)


if __name__ == '__main__':
    main()
```

This sort of structure is ideal for developing `argparse` CLIs.

Closing Comments

This article should serve as a pretty good primer for getting started with `argparse`. I have also included all the scripts used here in my GitHub script tip repo, if you're interested in playing around with them. That can be found here. Finally, I hope you enjoy making your own CLIs! The skill is ever a "tool for the box".