Sayak Paul
May 16th, 2019

PYTHON

Argument Parsing in Python

In this tutorial, learn how to parse one or more arguments from the command-line or terminal using the getopt, sys, and argparse modules.

If you plan a data science or a machine learning project, then it is not uncommon to get started developing it in a Jupyter Notebook. It provides interactivity to your computing tools, lets you run your modules quickly and so on. There are many pointers to choose Jupyter Notebooks as your primary weapon of choice, especially for doing data science or machine learning projects. However, this tutorial does not aim to shed light on that part.

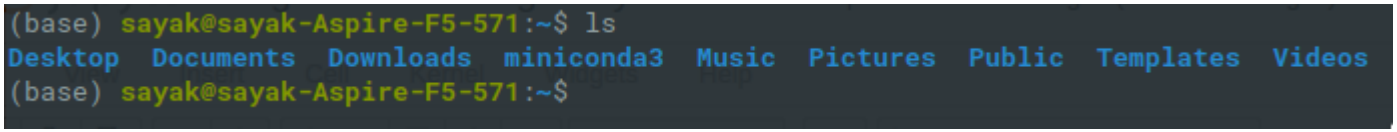
Consider you need to run a Python script as a batch job for the purpose of data ingestion in your data science project. Or you need to run a Python script to deploy a machine learning model to a remote server. In cases like these, just executing `python your_script.py` might not be sufficient. You may need to pass more *options* or *arguments* along with `python your_script.py`. You may already be familiar with the usage of additional arguments while running commands (like `ps`, `ls`, etc.) on a Linux terminal.

- A basic introduction to argument parsing
- Argument parsing in Python
  - Using `sys.argv`
  - Using `getopt`
  - Using `argparse`

Note that this tutorial assumes basic familiarity with Python.

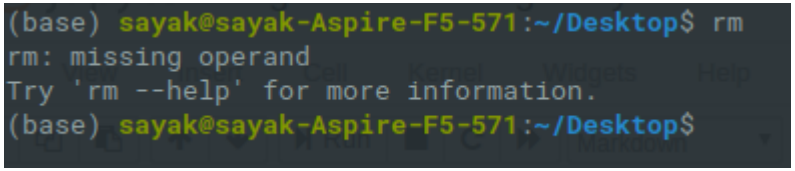
What is argument parsing?

If you are a Linux user, you might already be knowing this, I bet. In order to get a summary of the files and folders present in a particular directory, the command `ls` is often used. A typical output of running the `ls` command looks like -



However, you can supply many *options* to the `ls` command, `-l`, for example. Let's take a look at what that output looks like -

By passing the `-l` *option*, you got more information. Now, there is a slight difference between an *option* and an *argument* in this context. To remove a file or a folder from your current working directory, `rm` command is often used. Suppose, you have a text file named `demo.txt` and you ran `rm demo.txt` in order to remove the text file. In this case, `demo.txt` is the *argument* which you applied to the `rm` command. If you do not pass any *argument* to the `rm` command, you will get an error like so -



So, what is the difference between an *option* and an *argument*? It is now pretty obvious. Options are *optional* to pass whereas arguments are (often) *necessary* to pass. It is not essential to pass anything to the `ls` command for it to produce any output.

Now, when you hit `ls -l` or `rm demo.txt`, the operating system parses it in a certain way under the hood. This mechanism is generally specified by the developers of the operating system, and it is known as *parsing*.

So, that was a basic introduction to argument parsing. Almost all the programming languages come with support for argument parsing. Python is no exception to this. There are three very prevalent ways in which you can do argument parsing in Python -

- sys.argv
- getopt
- argparse

You will now take each of the above options one by one and see how to use them for parsing arguments (and options). Let's go chronologically.

Argument Parsing using sys.argv

Let's start simple. In this first example, you will create a Python file named `demo1.py`. Your program will accept an arbitrary number of arguments passed from the command-line (or terminal) while getting executed. The program will print out the arguments that were passed and the total number of arguments.

For example, if you execute `python demo1.py abc 123`, then the program would yield -

```
Number of arguments: 3
Argument(s) passed: ['demo1.py', 'abc', '123']
```

Now coming to the code –

```
import sys

print('Number of arguments: {}'.format(len(sys.argv)))
print('Argument(s) passed: {}'.format(str(sys.argv)))
```

You first imported the Python module `sys`, which comes with a standard installation of Python. You then employed the `argv` submodule which returns the list of the arguments passed to a Python script where `argv[0]` contains the name of the Python script. If you run `sys.argv` in a code cell like the following, you get the list of the configuration files responsible for making the IPython kernel function properly. It is better to not mess with them.

```
sys.argv
```

```
['/home/nbuser/anaconda3_501/lib/python3.6/site-packages/ipykernel/__main__.py',
 '-f',
 '/home/nbuser/.local/share/jupyter/runtime/kernel-dabba4f7-e3e0-4c39-99d2-261ba835c53f.json']
```

Let's now see how to use the `getopt` module for parsing arguments (and options).

Argument parsing using getopt

In comparison to `sys.argv`, the `getopt` module offers much more flexibility. Let's design a sample scenario first and write the code accordingly.

task of adding two numbers and provides an output. The only constraint is that the user needs to pass the inputs in the form of command-line arguments along with the Python script.

Talking a bit more practically, the script ideally should be executed like -

```
python add_numbers.py -a 3 -b 8
```

The output should be 11. Here, `-a` and `-b` are the options, and 3, 8 are the arguments that you provide to the script. The options not only enhance the readability part but also helps to decide the evaluation flow (consider if you are dividing instead of doing addition). Let's start looking at the code for this first.

```
import getopt
import sys

# Get the arguments from the command-line except the filename
argv = sys.argv[1:]
sum = 0

try:
    # Define the getopt parameters
    opts, args = getopt.getopt(argv, 'a:b:', ['foperand', 'soperand'])
    # Check if the options' length is 2 (can be enhanced)
    if len(opts) == 0 and len(opts) > 2:
      print ('usage: add.py -a <first_operand> -b <second_operand>')
    else:
      # Iterate the options and get the corresponding values
```

```
        print('Sum is {}'.format(sum))

except getopt.GetoptError:
    # Print something useful
    print ('usage: add.py -a <first_operand> -b <second_operand>')
    sys.exit(2)
```

The idea is to first get all the arguments using `sys.argv` and then process it accordingly. Let's now come to the most important line of code - `opts, args = getopt.getopt(argv, 'a:b:', ['foperand', 'soperand'])`

The signature of the `getopt()` method looks like:

`getopt.getopt(args, shortopts, longopts=[])`

- `args` is the list of arguments taken from the command-line.
- `shortopts` is where you specify the option letters. If you supply `a:`, then it means that your script should be supplied with the option `a` followed by a value as its argument. Technically, you can use any number of options here. When you pass these options from the command-line, they must be prepended with '-'.
- `longopts` is where you can specify the extended versions of the `shortopts`. They must be prepended with '--'.

You defined the `shortopts` to be `a:b:` which means your Python script would take two options as letters - 'a' and 'b'. By specifying ':' you are explicitly telling that these options will be followed by arguments.

The `getopt` module provides you with a handy Exception class `GetoptError` also for defining useful messages, so a to guide the user on how to use your Python script. This is why you wrapped the functional part of your script in a `try` block and defined the `except` block accordingly.

Here is how you can run the Python script:

```
(base) sayak@sayak-Aspire-F5-571:~/Desktop$ python add_numbers.py -a 8 -b 11
Sum is 19
(base) sayak@sayak-Aspire-F5-571:~/Desktop$ python add_numbers.py -a
usage: add.py -a <first_operand> -b <second_operand>
```

Note that the above script was *not* defined to handle the `longopts`. This is something you can try as an exercise.

Let's now see how to use `argparse` for argument parsing.

Argument parsing using argparse

From the above two options, it is quite viable that they are not very *readable*. Once you see `argparse` in action, you will also agree that the above two options lack on the flexibility part as well. To understand the usage of `argparse`, let's start with a code snippet which implements the scenario you saw in the above section.

```
import argparse

# Construct the argument parser
ap = argparse.ArgumentParser()
```

```
    help="first operand")
ap.add_argument("-b", "--soperand", required=True,
    help="second operand")
args = vars(ap.parse_args())

# Calculate the sum
print("Sum is {}".format(int(args['foperand']) + int(args['soperand'])))
```

First things first, `argparse` like the other two modules discussed above, comes with the standard installation of Python. You start by instantiating the `argparse` object. And the rest of the things become so simpler to write. Let's take the following line of code for example -

```
ap.add_argument("-a", "--foperand", required=True, help="first operand")
```

Here you added the argument that you expect to be supplied with the Python script when it is run. You provided the letter version of the argument along with its extended one. By specifying `required=True` you are explicitly asking the user to supply that particular argument. Finally, you appended a meaningful description of the argument which will be shown to the user in case he does not execute the script in the right manner.

The argument definition part is the same for the second argument, as well. You access the given arguments by specifying their respective indices.

The above code-snippet clearly shows how easy and flexible it is to define command-line argument parsing tasks with `argparse`. Here is how you can play with the above Python script:

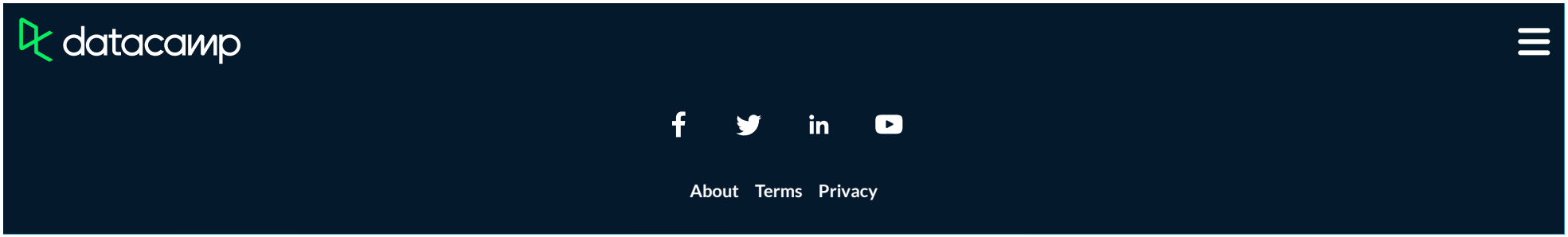
```
Sum is 26
(base) sayak@sayak-Aspire-F5-571:~/Desktop$ python argparse_d.py --help
usage: argparse_d.py [-h] -a FOPERAND -b SOPERAND

optional arguments:
  -h, --help            show this help message and exit
  -a FOPERAND, --foperand FOPERAND
                        first operand
  -b SOPERAND, --soperand SOPERAND
                        second operand
```

Notice how much easier the process became with `argparse`.

What's next?

Thank you for reading today's tutorial until the end. Now, you are equipped with the native Python modules that are able to parse arguments supplied from the command-line along with your Python script. As an exercise, you can customize your machine learning and data science projects accordingly to facilitate argument parsing and turn them into useful script utilities. Not just machine learning or data science, but anywhere you feel it is relevant enough. This is all for today's tutorial. If you are interested in enhancing your Python skills, you might want to take DataCamp's Python Data Science Toolbox (Part 1) course.


datacamp
About Terms Privacy