Parse Boolean Values from Command Line Arguments using Python

When executing Python scripts from any command line, for example, the Command Prompt, the Git Bash, the Linux Terminal, the MAC Shell, etc., we can pass arguments or values to them. These values are known as command line arguments. These arguments are generally integer numbers, floating numbers, strings, and boolean values represented as strings, which means that they have to be typecasted inside the code.

In this article, we will learn how to parse boolean values from command line arguments using Python.

Parse Boolean Values from Command Line Arguments using the `argparse` module

Python has a bunch of essential in-built modules such as `math`, `random`, `json`, `csv`, etc. that aim to solve general and repetitive programming problems. One such package is the `argparse` that helps programmers quickly write user-friendly command-line interfaces. Using this module, one can define the arguments that the program requires to work correctly. Apart from the definition, this module can also define default values, error messages, help texts, and perform common actions over values such as validation, typecasting, conversion, etc. We can use the `argparse` module for our use case.

We will implement a program that needs a string value and two boolean values to work properly. This is just an example for understanding purposes.

Refer to the following Python code for the same.

```
import argparse

def parse_boolean(value):
    value = value.lower()

    if value in ["true", "yes", "y", "1", "t"]:
        return True
    elif value in ["false", "no", "n", "0", "f"]:
        return False

    return False

def output(name, burger, cake):
    if burger and cake:
        return f"{name} is eating a burger and a cake."
    elif burger:
        return f"{name} is eating a burger."
    elif cake:
        return f"{name} is eating a cake."

    return f"{name} is eating nothing."

parser = argparse.ArgumentParser(description = "A program that accepts one string
```

```
and two boolean values.")
parser.add_argument("name", help = "Name of a person.")
parser.add_argument("--burger", type = parse_boolean, default = False, help =
"Flag for burger.")
parser.add_argument("--cake", type = parse_boolean, default = False, help = "Flag
for cake.")
args = parser.parse_args()
print(output(args.name, args.burger, args.cake))
```

Python Code Explanation

The Python code above first creates an object of the `ArgumentParser` class. This class has all the utilities for parsing the command line arguments. Following is the class signature of this class.

```
class argparse.ArgumentParser(prog=None, usage=None, description=None,
epilog=None, parents=[], formatter_class=argparse.HelpFormatter, prefix_chars='-',
fromfile_prefix_chars=None, argument_default=None, conflict_handler='error',
add_help=True, allow_abbrev=True, exit_on_error=True)
```

All these arguments are keyword arguments, and they should be passed as keyword arguments to the class.

Next, using the `add_argument()` function of this object, we add arguments to the parser. Following are some of the main arguments for this function.

- `name` - A name or a list of option strings. For example, `world`, or `--world` and `-w`.
- `action` - The action to take when this argument is found in the command line arguments.
- `nargs` - The number of command line arguments that should be considered.
- `type` - The data type to which the argument should be converted to.
- `const` - A constant value required for `action` and `nargs`.
- `default` - The default value that should be considered if the argument is missing from the command line arguments.
- `help` - The help text for the argument.

We add three arguments, namely, `name`, `--burger`, and `--cake`. Here, `name` is a positional argument and `--burger` and `--cake` are two optional arguments. For each argument, we define the help text, and for optional arguments, we also define the default values.

Next, we convert the string arguments to their respective objects or data types using the `parse_args()` method. This method returns a `Namespace` object that contains all the arguments as keys pointing to their respective converted values. We can access these typecasted values as follows.

```
args.name
args.burger
args.cake
```

We also define two functions, namely, `parse_boolean()` and `output()`. The `parse_boolean()` function accepts the value entered as a command line argument next to optional boolean arguments. Then it interprets the value and returns the corresponding boolean value. Note that here the function will return `False` for any gibberish value. One can also raise an `Exception` instead of returning some default value. You can test that using the following command.

```
python main.py Vaibhav --burger Hello --cake World
```

Output:

```
Vaibhav is eating nothing.
```

The other function is the `output()` function. This function accepts the three values (one string and two booleans) and returns a suitable output string.

Testing

To test the program, execute the following commands from any command line. Note that the following commands consider that the above Python code is present inside a file by the name of `main.py`.

```
python main.py Vaibhav --burger True --cake True
python main.py Vaibhav --burger T --cake 0
python main.py Vaibhav --burger False --cake False
python main.py Vaibhav --burger True --cake False
python main.py Vaibhav --burger False --cake True
python main.py Vaibhav --burger ABC --cake XYZ
python main.py Vaibhav
python main.py Vaibhav --burger True
python main.py Vaibhav --cake False
python main.py
```

Output:

```
Vaibhav is eating a burger and a cake.
Vaibhav is eating a burger.
Vaibhav is eating nothing.
Vaibhav is eating a burger.
Vaibhav is eating a cake.
Vaibhav is eating nothing.
Vaibhav is eating nothing.
Vaibhav is eating a burger.
Vaibhav is eating nothing.
usage: main.py [-h] [--burger BURGER] [--cake CAKE] name
main.py: error: the following arguments are required: name
```

As we can see, the last command, `python main.py`, throws an error. The reason behind the same is simple; `name` is a positional and compulsory argument.