Theory: Argparse module

27 minutes 0 / 5 problems solved

If you're writing a user-friendly program, one way to make it more universal is to use the command line and let users specify all the necessary parameters and their values themselves. By doing that, you design a program capable of taking different numbers if it's a calculator, for example, or a path to a file, as it's often required, without making a user go inside the script trying to find where and what should be replaced.

The `argparse` is one of the modules that lets you do that. It allows you to pass the arguments through the command line and also assign names to them, use them as "flags", automatically generate messages for users, and a lot of other cool things we will get to a bit later.

We will write a script called `recipe_book.py` as an example that takes up to five ingredients and prints a recipe of a dish you can cook with the provided ingredients.

§1. Getting started with argparse

The first thing is to import the module:

```
1 import argparse
```

The next step is to create an `ArgumentParser` object which will store all the information about the arguments:

```
1
parser = argparse.ArgumentParser(description="This program prints recipes \
2 consisting of the ingredients you provide.")
```

The `ArgumentParser` has quite a number of parameters that you can specify, but we only invoked `description` which is quite handy in order to explain to a user what your program is for in general. Now let's add some arguments.

§2. Adding arguments

To do that, we will use the `add_argument()` method:

```
1 parser.add_argument("-i1", "--ingredient_1")  # optional argument
2                                               # or
3 parser.add_argument("ingredient_1")           # positional argument
```

We also need to note the difference between the optional and the positional arguments. When parsing, if an argument has a dash `-` or a double dash `--` prefix, it'll be treated as an optional. Let's take a closer look at the first line of the code in the example above. With *optional* arguments, traditionally, a single dash `-` denotes a short version of a name (usually consists of only one letter), while a double dash `--` is used for a full argument name. When specifying this argument from the command line, you can use either of these variants. Since *positional* arguments are used without a prefix before them, they can have only one name.

The `add_argument()` has a lot of useful parameters, but we are going to look at the most commonly-used ones. For example, the parameter "action" is responsible for what should be done with a command-line argument. By default, it just stores the value passed to the argument, though it's not the only option.

```
1 parser.add_argument("--salt", action="store_true")
```

Since pretty much everybody has some salt in their kitchen, we'll assume that our user has it by default. So, we'll use it as a flag instead of storing it as a stand-alone ingredient. In the example above, we have done so by setting

Current topic:

Argparse module

Topic depends on:

× Command line arguments

Table of contents:

the action to the "`store_true`". It is used to assign boolean values to the corresponding arguments, so that the `salt` value will be `False` by default, but if the user lists `--salt` among the arguments, the value will be changed to `True`. There's also an opposite option, `store_false`: it sets the corresponding value to `True` by default, but makes it `False` if the argument is listed.

The same can be achieved by specifying the `default` parameter:

```
parser.add_argument("--pepper", default=False)
```

This time the argument isn't used as a flag any more, so, if you'd like to change the value, you will have to define it in the command line explicitly: `--pepper "True"`.

Finally, since we're only at the beginning of the development process of our program, it might be useful to limit the choice of each ingredient to only those used in our recipes. This can be done with the `choices` parameter that will show the options as a value for a particular argument:

```
parser.add_argument("-i2", "--ingredient_2",
                    choices=["pasta", "rice", "potato", "onion",
                     "garlic", "carrot", "soy_sauce", "tomato_sauce"],
                help="You need to choose only one ingredient from the list.")
```

Another useful parameter you see here is `help`. It contains a brief description of an argument and also allows you to guide a user in their work with a script.

§3. Parsing arguments

The `parse_args()` method is used for reading argument strings from the command line:

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

Now we can access the values specified by a user as attributes of the `args`. The long versions are used as attribute names:

```
print(args.ingredient_2)  # onion
# (the value was chosen by a user from the given options)
```

Note that we can't use short versions of arguments: for example, `args.i2` **will not** work.

In case a user didn't specify an optional argument in the command line, the value is set to `None` by default:

```
print(args.ingredient_3)  # None
# (the value wasn't provided by a user)
```

So far the code of our program in the "recipe_book.py" module looks as follows:

```
import argparse



parser = argparse.ArgumentParser(description="This program prints recipes \
consisting of the ingredients you provide.")

parser.add_argument("-i1", "--ingredient_1", choices=
["pasta", "rice", "potato",

                    "onion", "garlic", "carrot", "soy_sauce", "tomato_sauce"],

                    help="You need to choose only one ingredient form the list.")

parser.add_argument("-i2", "--ingredient_2", choices=
["pasta", "rice", "potato",


                    "onion", "garlic", "carrot", "soy_sauce", "tomato_sauce"],


                    help="You need to choose only one ingredient form the list.")

parser.add_argument("-i3", "--ingredient_3", choices=
["pasta", "rice", "potato",


                    "onion", "garlic", "carrot", "soy_sauce", "tomato_sauce"],


                    help="You need to choose only one ingredient form the list.")

parser.add_argument("-i4", "--ingredient_4", choices=
["pasta", "rice", "potato",


                    "onion", "garlic", "carrot", "soy_sauce", "tomato_sauce"],


                    help="You need to choose only one ingredient form the list.")

parser.add_argument("-i5", "--ingredient_5", choices=
["pasta", "rice", "potato",


                    "onion", "garlic", "carrot", "soy_sauce", "tomato_sauce"],


                    help="You need to choose only one ingredient form the list.")



parser.add_argument("--salt", action="store_true",


                help="Specify if you'd like to use salt in your recipe.")

parser.add_argument("--pepper", default="False",


                help="Change to 'True' if you'd like to use pepper in your rec
ipe.")



args = parser.parse_args()



ingredients = [args.ingredient_1, args.ingredient_2, args.ingredient_3,

                args.ingredient_4, args.ingredient_5]

if args.salt:

    ingredients.append("salt")

if args.pepper == "True":
```

```
    ingredients.append("pepper")


print(f"The ingredients you provided are: {ingredients}")



def find_a_recipe(ingredients):
    ...

# processes the input and returns a recipe depending on the provided ingredien
ts
```

§4. How do you actually use that in the command line?

Now let's see how it looks like from the user perspective. Here's a sample call of our program from the command line:

```
python recipe_book.py -i1 rice -i2 onion -i3 garlic -i4 carrot -
i5 tomato_sauce --salt

# The ingredients you provided are: ['rice', 'onion', 'garlic', 'carrot', 'tomato_
sauce', 'salt']
# <The description of the available recipe>
```

What's important to note here is that the format `argument value` and `argument=value` are equivalent:

```
python recipe_book.py -i1=pasta -i2=garlic -i3=tomato_sauce --salt --
pepper="True"

# The ingredients you provided are: ['pasta', 'garlic', 'tomato_sauce', None, None
, 'salt', 'pepper']
# <The description of the available recipe>
```

However, if a user tries to use an option which is not given in the `choices` parameter, it will raise an error:

```
python recipe_book.py -i1 bread -i2 onion -i3 garlic -i4 carrot -
i5 tomato_sauce --salt
# usage: recipe_book.py [-h]
#                       [-
i1 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i2 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i3 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i4 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i5 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [--salt] [--pepper PEPPER]
# recipe_book.py: error: argument -i1/--
ingredient_1: invalid choice: 'bread'


# (choose from 'pasta', 'rice', 'potato', 'onion', 'garlic', 'carrot', 'soy_sauce'
, 'tomato_sauce')
```

Note that the first thing displayed is the 'usage' of our program. We did not specify it ourselves when creating the argument parser, so it was generated automatically from the parser's arguments. In the 'usage', we can see that the value 'bread' is not supported by our program, and the error message also explains this fact.

Remember the `help` parameter we discussed earlier? When a user specifies `--help` or `-h` as an argument in the command line, the description for each argument is displayed:

```
python recipe_book.py --help
# usage: recipe_book.py [-h]
#                       [-
i1 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i2 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i3 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i4 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [-
i5 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}]
#                       [--salt] [--pepper PEPPER]
#

# This program prints recipes consisting of the ingredients you provide.

#

# optional arguments:

#   -h, --help            show this help message and exit

#   -i1 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}, --
ingredient_1

# {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}


#                       You need to choose only one ingredient from the list.

#   -i2 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}, --
ingredient_2

# {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}


#                       You need to choose only one ingredient from the list.

#   -i3 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}, --
ingredient_3

# {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}


#                       You need to choose only one ingredient from the list.

#   -i4 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}, --
ingredient_4

# {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}


#                       You need to choose only one ingredient from the list.

#   -i5 {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}, --
ingredient_5

# {pasta,rice,potato,onion,garlic,carrot,soy_sauce,tomato_sauce}


#                       You need to choose only one ingredient from the list.

#   --
salt                Specify if you'd like to use salt in your recipe.

#   --
pepper              Change to 'True' if you'd like to use pepper in your recipe.
```

Again, first we see the 'usage' of our program, then, there's the description we wrote, and, finally, the list of all arguments.

§5. Summary

In this topic, we briefly familiarized ourselves with Python `argparse` module. There are three main steps to get the job done: first, create the `ArgumentParser` object; then, add arguments with `add_argument()` method; finally, parse them by `parse_args()` method and use in your program. Since what we discussed here is more of a review than a full description, it's definitely worth reading the official docs, argparse section for more details, especially to learn about different parameter options you can use in your program.

Report a typo

70 users liked this theory. **9** didn't like it. **What about you?**

Start practicing

Comments (6) Hints (0) Useful links (0) Show discussion