Create an argparse interface

Estimated Time: 1-2 hours

So far you have seen how to make simple command line interfaces for applications using input() and sys.argv to obtain command line arguments. When you want to build more complex interfaces (think of the myriad options you can pass to the git command, for example), using sys.argv quickly becomes clunky.

In this assignment you will learn about a nicer way to handle more complex interfaces using the argparse module, using it to build the interface to our command line code snippets app.

Planning the interface

To create a command line interface you first need to decide how you want it to look. The `put` function takes two arguments, so it makes sense to allow two command line arguments. Since you want to retrieve and set snippets, it also makes sense to add an initial argument specifying the command type (storage or retrieval). Next, you'll sketch out a few ways you can use the app.

```
# Store an item using fully qualified names
python3 snippets.py --type "put" --name "list" --snippet "A sequence o

# Store an item using abbreviations
python3 snippets.py -t "put" -n "list" -s "A sequence of things - crea
```

```
# Use positional rather than optional arguments
python3 snippets.py put list "A sequence of things - created using []"
```

The options above demonstrate the two type of arguments which argparse can handle. The first two examples use **option arguments**. These are arguments which can be provided in any order. So for example we could supply the snippet, then the type, then the name. The third example uses **positional arguments**. Here the positions are fixed, so the arguments have to be provided in a set order.

Each option presented above has advantages and disadvantages. For an app as simple as this, using optional arguments everywhere is too verbose, even when abbreviated. For this assignment the last option is ideal syntax.

Now that you have settled on an interface, you'll start to build the parser using the `argparse` module. Place this into a new main() function in *snippets.py*. This will ensure that the function only executes when you run the module as a program, and will not execute when the module is imported.

```
def main():
    """Main function"""
    logging.info("Constructing parser")
    parser = argparse.ArgumentParser(description="Store and retrieve s
    arguments = parser.parse_args()

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

Here you create an instance of the `ArgumentParser` object from the `argparse` module. Note that you will need to add `import argparse` at the top of your script to access this object. You will use this object to describe your interface and parse the list of arguments (i.e. to split it up according to the interface you have described).

Next you'll call the `parse_args` method of the parser, letting it locate the arguments automatically. As you learned in Unit 1, the first argument of `sys.argv` contains the name of our program; argparse is smart enough to know not to parse that as an argument, but to use it if it needs to print out usage help.

The `parse_args` function takes the list of arguments (for example `put list "A sequence of things - created using []"`), and splits them into named variables. These variables can be accessed using the `arguments` object, for instance, we can create the variables arguments.name and arguments.snippet.

By this point you have added a main block to the app that calls our main function. This means that you can try running our script from the command line. Open up a terminal, navigate to your project folder, and enter `python3 snippets.py --help`. You should see some help text the parser object automatically generated.

Before moving on, be sure to commit your changes.

Making the interface accept arguments

Now that you have a parser you can start building up the interface. As a first step you're going to split the parser into subparsers – one for the storage command and one for the retrieval command. This is similar to how apps like `git` work, with different commands taking different arguments.

```
def main():
    """Main function"""
    logging.info("Constructing parser")
    parser = argparse.ArgumentParser(description="Store and retrieve s
    subparsers = parser.add_subparsers(dest="command", help="Available

    # Subparser for the put command
    logging.debug("Constructing put subparser")
    put_parser = subparsers.add_parser("put", help="Store a snippet")

    arguments = parser.parse_args()
```

Here you're using the `add_subparsers` method of the parser to add multiple subparsers. Then you're using the `add_parser` method of the `subparsers` object to create the new subparser. This subparser works in the same way as a standard `ArgumentParser` instance, allowing you to add arguments. You set the `dest` argument to the string `"command"` . This will allow you to access the command name (for example `"put"` or `"get"`) through the `arguments.command` variable. Next, try to expand our function even further to add in the arguments for a put command:

```
def main():
    """Main function"""
    logging.info("Constructing parser")
    parser = argparse.ArgumentParser(description="Store and retrieve s

    subparsers = parser.add_subparsers(dest="command", help="Available

    # Subparser for the put command
    logging.debug("Constructing put subparser")
    put_parser = subparsers.add_parser("put", help="Store a snippet")
    put_parser.add_argument("name", help="Name of the snippet")
    put_parser.add_argument("snippet", help="Snippet text")

    arguments = parser.parse_args()
```

Here you'll use the `add_argument` method to add the two positional arguments. Each one has its own explanatory text, which you'll see in a moment.

Now try it out. In a terminal, run the program a few different ways to check that the parser works:

```
# Shows the help for which subcommands are available
python3 snippets.py --help

# Show the help for the put subcommand
python3 snippets.py put --help

# Standard put command
python3 snippets.py put list "A sequence of things - created using []"

# Put command missing an argument (this should fail)
python3 snippets.py put list
```

You've reached another milestone, so be sure to commit these changes.

Argument parsing challenge

Create the `get` subparser for your program. This should be similar to the `put` subparser, except that it will only take one argument – the name of the snippet.

Finishing touches

Now that you have both the functionality and interface, all that remains is to join the two together. Extend the `main` function:

```
def main():

    ...

    arguments = parser.parse_args()
    # Convert parsed arguments from Namespace to dictionary
    arguments = vars(arguments)
    command = arguments.pop("command")

    if command == "put":
        name, snippet = put(**arguments)
        print("Stored {!r} as {!r}".format(snippet, name))
    elif command == "get":
        snippet = get(**arguments)
        print("Retrieved snippet: {!r}".format(snippet))
```

There are a few things which might be new to you in this code. Firstly, you're using the built in `vars` function to convert the `Namespace` object returned by `parse_args` into a dictionary. This is purely for convenience, as the dictionary object is somewhat easier to work with.

Next, you'll see the `pop` method of the dictionary. This removes and returns the `command` item which you added earlier to distinguish between the two commands.

You then call the `put` or get functions, using `**arguments` as an argument. Using the double-star operator in this context is known as *unpacking*. It converts the key-value pairs in the dictionary into keyword arguments to the function. For example:

```
# Code without argument unpacking
put(name="list", snippet="A sequence of things - created using []")

# Identical code which uses argument unpacking
arguments = {
    "name": "list",
    "snippet": "A sequence of things - created using []"
}
put(**arguments)
```

Try running your app and using it to try to add and retrieve items. Open up the log to make sure that the FIXME lines are being shown. If so, congratulations – that's the first part of the app finished.

Don't forget to commit your changes into git with an appropriate message, for example: `git commit -m 'Add parser structure'` .

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

✓ Mark as completed

Previous

Next