ChainMap

A **ChainMap** is a class that provides the ability to link multiple mappings together such that they end up being a single unit. If you look at the documentation, you will notice that it accepts **maps*, which means that a ChainMap will accept any number of mappings or dictionaries and turn them into a single view that you can update. Let's look at an example so you can see how this works:

```
from collections import ChainMap
car_parts = {'hood': 500, 'engine': 5000, 'front_door': 750}
car_options = {'A/C': 1000, 'Turbo': 2500, 'rollbar': 300}
car_accessories = {'cover': 100, 'hood_ornament': 150, 'seat_cover': 99}
car_pricing = ChainMap(car_accessories, car_options, car_parts)
print (car_pricing['hood'])
#500
```

Here we import **ChainMap** from our collections module. Next we create three dictionaries. Then we create an instance of our ChainMap by passing in the three dictionaries that we just created. Finally, we try accessing one of the keys in our ChainMap. When we do this, the ChainMap will go through each map in order to see if that key exists and has a value. If it does, then the ChainMap will return the first value it finds that matches that key.

This is especially useful if you want to set up defaults. Let's pretend that we want to create an application that has some defaults. The application will also be aware of the operating system's environment variables. If there is an environment variable that matches one of the keys that we are defaulting to in our application, the environment will override our default. Let's further pretend that we can pass arguments to our application. These arguments take precendence over the environment and the defaults. This is one place where a ChainMap can really shine. Let's look at a simple example that's based on one from Python’s documentation:

```
import argparse
import os

from collections import ChainMap


def main():
    app_defaults = {'username':'admin', 'password':'admin'}

    parser = argparse.ArgumentParser()
    parser.add_argument('-u', '--username')
    parser.add_argument('-p', '--password')
    args = parser.parse_args()
    command_line_arguments = {key:value for key, value
                              in vars(args).items() if value}

    chain = ChainMap(command_line_arguments, os.environ,
                     app_defaults)
    print(chain['username'])

if __name__ == '__main__':
    main()
    os.environ['username'] = 'test'
    main()
```

Let’s break this down a little. Here we import Python’s **argparse** module along with the **os** module. We also import ChainMap.Next we have a simple function that has some silly defaults. I’ve seen these defaults used for some popular routers. Then we set up our argument parser and tell it how to handle certain command line options. You will notice that argparse doesn’t provide a way to get a dictionary object of its arguments, so we use a dict comprehension to extract what we need. The other cool piece here is the use of Python’s built-in **vars**. If you were to call it without an argument, vars would behave like Python’s built-in **locals**. But if you do pass in an object, then vars is the equivalent to object’s **__dict__** property.

In other words, **vars(args)** equals **args.__dict__**. Finally create our ChainMap by passing in our command line arguments (if there are any), then the environment variables and finally the defaults. At the end of the code, we try calling our function, then setting an environment variable and calling it again. Give it a try and you’ll see that it prints out **admin** and then **test** as expected. Now let’s try calling the script with a command line argument:

```
python chain_map.py -u mike
```

When I ran this, I got **mike** back twice. This is because our command line argument overrides everything else. It doesn’t matter that we set the environment because our ChainMap will look at the command line arguments first before anything else.

Now that you know how to use ChainMaps, we can move on to the Counter!