5.1 The configparser module

Introduction to the configparser module

Currently, many popular services provide an API that we can use in our applications. Integration with these services requires authentication using data such as a login and password, or simply an access token.

Each service may require different data for authentication, but one thing is certain – we need to store it somewhere in our application. It's not a good idea to hardcode them directly in the code.

A better solution is to use the configuration file, which will be read by the code. In Python, this is possible thanks to a module called `configparser`.

The `configparser` module is available in the Python standard library. To start using it, we need to import the appropriate module:

```
import configparser
```

However, before we start reading the configuration data, we must familiarize ourselves with the structure of the file in which they're stored. Are you curious how this is done?

What does the configuration file look like?

The structure of the configuration file is very similar to Microsoft Windows INI files. It consists of sections that are identified by names enclosed in square brackets. The sections contain items consisting of key value pairs. Each pair is separated by a colon `:` or equals sign =.

What's more, the configuration file may contain comments followed by a semicolon `;` or hash #:

```
[DEFAULT]
host = localhost # This is a comment.

[mariadb]
name = hello
user = user
password = password

[redis]
port = 6379
db = 0
```

Our configuration file contains the DEFAULT, `mariadb` and `redis` sections.

The DEFAULT section is slightly different because it contains the default values that can be read in the other sections of the file. In our case, there's a common host for all sections.

The second section called `mariadb` stores the data necessary to connect to the MariaDB database. These are the database name, username, and password.

The last section contains `Redis` configuration data, consisting of the port and database number. In addition, both in this section and in the `mariadb` section, we have access to the host option defined in the DEFAULT section. In a moment, you'll learn how to read this data using the `configparser` module.

NOTE: The whitespace at the beginning and end of keys and values is removed.

Parsing the configuration file

Parsing the configuration file is extremely simple. First, we need to create a `ConfigParser` object, which provides many useful methods for parsing data. One of them is the `read` method, responsible for reading and parsing the configuration file. In our example, we pass the `config.ini` filename to it, but it's also possible to pass a list containing several files.

If all goes well, the `read` method returns a list of filenames that have been successfully parsed. Let's look at the code in the editor to see how to parse the data stored in the `config.ini` file.

```
import configparser

config = configparser.ConfigParser()
print(config.read('config.ini'))

print('Sections:', config.sections(),'\n')

print('mariadb section:')
print('Host:', config['mariadb']['host'])
print('Database:', config['mariadb']['name'])
print('Username:', config['mariadb']['user'])
print('Password:', config['mariadb']['password'], '\n')

print('redis section:')
print('Host:', config['redis']['host'])
print('Port:', int(config['redis']['port']))
print('Database number:', int(config['redis']['db']))
```

Result:

```
Sections: ['mariadb', 'redis']

mariadb section:
Host: localhost
Database: hello
Username: user
Password: password

redis section:
Host: localhost
Port: 6379
Database number: 0
```

In the example, we use the `sections` method to display the names of sections in the file. Note that the `DEFAULT` section doesn't appear in the list of returned sections. This is the default behavior of the `sections` method.

Access to the data contained in the configuration file is analogous to how we use dictionaries. To obtain any value, use the appropriate key sequence. It's important to note that the section names are case sensitive, while the keys aren't.

Despite the fact that the `DEFAULT` section is omitted as a result of the `sections` method, we still have access

to its options. Both the `mariadb` and `redis` sections can read the `host` option.

Its also possible to access the values stored in the options by using the `get` method. The `get` method requires the section name and key to be passed. This is what it looks like in practice:

```
print('Host:', config.get('mariadb', 'host')) # print('Host:',
config['mariadb']['host']
```

Reading configuration from other sources

The `configparser` module enables configurations from various sources to be read. One of them is a dictionary that we can load using the `read_dict`. Look at the code in the editor.

```
import configparser

config = configparser.ConfigParser()

dict = {
    'DEFAULT': {
        'host': 'localhost'
    },
    'mariadb': {
        'name': 'hello',
        'user': 'root',
        'password': 'password'
    },
    'redis': {
        'port': 6379,
        'db': 0
    }
}

config.read_dict(dict)

print('Sections:', config.sections(),'\n')

print('mariadb section:')
print('Host:', config['mariadb']['host'])
print('Database:', config['mariadb']['name'])
print('Username:', config['mariadb']['user'])
print('Password:', config['mariadb']['password'], '\n')

print('redis section:')
print('Host:', config['redis']['host'])
print('Port:', int(config['redis']['port']))
print('Database number:', int(config['redis']['db']))
```

Result:

```
Sections: ['mariadb', 'redis']

mariadb section:
Host: localhost
Database: hello
```

```
Username: root
Password: password

redis section:
Host: localhost
Port: 6379
Database number: 0
```

The `read_dict` method accepts any dictionary whose keys are section names, while the values include dictionaries containing keys and values. All values read from the dictionary are converted to strings.

NOTE: The `configparser` module also has `read_file` and `read_string` methods that allow you to read the configuration from an open file or string. You can find more information about these methods in the documentation.

Creating a configuration file

Creating a configuration file is as easy as parsing it. If you know how to work with dictionaries, it's a piece of cake. Let's look at a simple example of how to create a configuration file that you already know. Take a look at the code in the editor.

```
import configparser

config = configparser.ConfigParser()

config['DEFAULT'] = {'host': 'localhost'}
config['mariadb'] = {'name': 'hello',
                     'user': 'root',
                     'password': 'password'}
config['redis'] = {'port': 6379,
                   'db': 0}

with open('config.ini', 'w') as configfile:
    config.write(configfile)
```

To create a configuration file, you should treat the `ConfigParser` object as a dictionary. Note that the section names are keys, while their options are listed in separate dictionaries. The above configuration is saved using the `write` method, which requires an open file to be passed in text mode. For this purpose, the built-in open method is used.

A configuration loaded using the read method can also be modified. To change a single option, simply set the new value to the appropriate key, and then save the file using the write method:

```
import configparser

config = configparser.ConfigParser()
config.read('config.ini')

config['redis']['db'] = '1'

with open('config.ini', 'w') as configfile:
    config.write(configfile)
```

Interpolating values

The big advantage of the configuration file is the possibility of using interpolation. It allows you to create expressions consisting of a placeholder under which the appropriate value will be substituted. Take a look at the configuration file below:

```
[DEFAULT]
host = localhost

[mariadb]
name = hello
user = user
password = password

[redis]
port = 6379
db = 0
dsn = redis://%(host)s
```

The configuration file has been extended with another option called `dsn`. Its value contains the placeholder `%(host)s`, which needs to be replaced by an appropriate value.

Placing any key between `%` and `s` informs the parser of the need to interpolate. Of course, all the work is done for us, and we only get the ready results.

For the `dsn` option, it'll be the following string: `redis://localhost`. Note that the placeholder `%(host)s` has been replaced by the value stored in the host option.