How To Work With Config Files In Python

Working with INI-files in Python using the built-in configparser module - ***by Florian Dahlitz on November 09, 2020 (10 min)***

Table of Contents

Introduction

For every developer the day comes, when he or she is working on large and consequently complex projects. To maintain the software, configuration files can be very beneficial and time-saving. Instead of changing the source code itself, "only" configuration files need to be adjusted or exchanged to access a different API endpoint, update the base URL, or similar things.

While there are various ways to support configuration files in your software including JSON, YAML, and plain-text files, this article aims to give you an introduction to the *configparser* module from the standard library.

Note: This article is based on Python 3.9.0 (CPython). The source code can be found on GitHub.

File structure

Before jumping into the code, let's have a look at how an actual configuration file can look like.

```
[DEFAULT]
admin_page = no
landing_page = yes
moderator_page = no  # looks good here

[moderator]
moderator_page = yes

[admin]
admin_page = yes
moderator_page = yes
```

In the example at hand, we have a configuration file called *config.ini*, which consists of three sections. Each section consists of a section title, which is encapsulated within square brackets, and a list of key-value pairs. Notice that comments are supported, too. Everything after a `#` (number sign, hash) or `;` (semicolon) will be ignored.

While the *moderator* and *admin* sections are simply collections of key-value pairs, the *DEFAULT* section (first section) is somewhat special. It contains default values if one of the other sections does not provide a value for a certain key. Consequently, if you try to access a value in one of the other sections but the key is not present, the parser returns the value from the default section (if present) instead of raising a `KeyError`. More on that later.

Let's finalise the story of our configuration file. For this scenario, we manage a user's page access through this configuration file. Therefore, the default section represents the permissions for a normal user, whereas the moderator and admin sections contain the permissions for moderators and administrators respectively.

Accessing the file's content

The `ConfigParser` object is the main configuration parser and the main object of the *configparser* module. You could implement your own configuration parser using the mapping protocol, but let's stick to the `ConfigParser` in this article.

> **Note:** There is also a legacy API available with explicit `get()` and `set()` methods. However, it is highly preferred to stick to the `ConfigParser` class or to implement a custom parser based on the mapping protocol.

While there are a bunch of parameters, which `ConfigParser` accepts, we will stick to the default values in this article. However, if you are looking for more customisation, make sure to check out the respective part of the documentation [1].

Let's create a new file in our working directory alongside the *config.ini* file called *parser_playground.py*. First of all, we import the `ConfigParser` class from the `configparser` module and create an instance of that class.

```
# parser_playground.py
from configparser import ConfigParser

config = ConfigParser()
```

Our `config` object does not contain any information, yet. To change that, we need to read the *config.ini* file first. This can be done by calling the `read()` method of the `ConfigParser` instance (here `config`).

```
# previous code in parser_playground.py
config.read("config.ini")
```

Notice, that the `read()` method also accepts a list of path-like objects [2], which it will read from. After reading the configuration file, we can start exploring how to access information stored in it. Let's start by having a look at how to handle sections. First, we want to list all available sections. This can be achieved by using the `ConfigParser`'s `sections()` method:

```
# previous code in parser_playground.py
print(f"Sections: {config.sections()}")  # Sections: ['moderator', 'admin']
```

Furthermore, we can explicitly check, whether a certain section exists by using the parser's `has_section()` method:

```
# previous code in parser_playground.py
print(f'Does a section called "admin" exist: {config.has_section("admin")}')  # True
print(f'Does a section called "user" exist: {config.has_section("user")}')  # False
print(f'Does a section called "DEFAULT" exist: {config.has_section("DEFAULT")}')  # False
```

> **Note:** The default section is neither listed when calling the `sections()` method nor is it acknowledged by the `has_section()` method.

Next, we want to access individual values. But before accessing a specific value using its identifier, we can list all available options of one section using the `options()` method and supplying the section name as an argument:

```
# Previous code in parser_playground.py
print(f'Options: {config.options("admin")}')  # Options: ['admin_page', 'moderator_page', 'landing_page']
```

Additionally, we can utilise the `has_option()` method to check whether a given section includes a certain option:

```
# Previous code in parser_playground.py
print(f'"admin_page" in "admin" section: {config.has_option("admin", "admin_page")}')
```

To access the values of a section, you can use the parser's `get()` method and supply a section name and an option name. The values will always be strings if present. If you need them in another format, consider using the respective `getboolean()`, `getint()`, and `getfloat()` methods. They will try to parse the strings to the desired data type.

To conclude this section, *Mapping Protocol Access* needs to be mentioned. This generic name means that values can be accessed as if we were dealing with a dictionary. Namely, we can use the `config["section"]["option"]` notation to access a certain value or even check if a certain option is present in a section:

```
# Previous code in parser_playground.py
print("admin_page" in config["admin"])  # True
print(config["admin"]["admin_page"])  # yes
```

Modifying information

Next, let's have a look at how to add or change information and write it back to the configuration file. Again, we start with sections. To add a section, we can use the `ConfigParser`'s `add_section()` method. It accepts a section name as a string and adds the respective section to the parser. Supplying a different data type results in a `TypeError`. If the section already exists, a `DuplicateSectionError` is raised. Trying to name the section `default` results in a `ValueError`.

```
# Previous code in parser_playground.py
config.add_section("unknown")
print(f'Sections: {config.sections()}')  # Sections: ['moderator', 'admin', 'unknown']
```

To delete a section, simply use the `remove_section()` method.

```
# Previous code in parser_playground.py
config.remove_section("unknown")
print(f'Sections: {config.sections()}')  # Sections: ['moderator', 'admin']
```

Python's `ConfigParser` object provides similar methods for manipulating options. For instance, the `set()` method can be invoked to not only add new options to a section but update existing options as well. Likewise, if you want to delete a certain option entirely, use the parser's `remove_option()` method.

parser's remove_option() method.

```
# Previous code in parser_playground.py
config.set("admin", "admin_page", "false")
config.remove_option("admin", "moderator_page")
print(f'Options in "admin" section: {config.items("admin")}')
```

After manipulating the configuration, we can write it back to the same or a different file as follows:

```
# Previous code in parser_playground.py
with open("config1.ini", "w") as f:
    config.write(f)
```

Notice, that the `write()` method accepts a file object [3], which is opened in text mode (accepting string, not bytes). This closes the section about manipulating information of the configuration read earlier from *config.ini*.

Interpolation

Last but not least, let's have a look at something making `ConfigParser` superior to Python's *json* module (at least in my opinion): **Interpolation**. Interpolation means that values can be pre-processed before they are returned by calls of some `get()` method. The `configparser` module provides two interpolation classes: `BasicInterpolation` and `ExtendedInterpolation`. The first one only allows reusing options from the same section within the configuration file and its syntax is not as pretty as the one from the latter class. That is why we keep things simple at this point and only have a look at the `ExtendedInterpolation` class.

The following snippet shows you a configuration file making use of extended interpolation syntax.

```
# interpolation_config.ini
[paths]
root_dir = /home/florian
downloads_dir = ${root_dir}/Downloads

[destinations]
app_dir = ${paths:downloads_dir}/application/python
```

In essence, the first section defines the path to the root directory. This path is used as a prefix for the second option, the path to the downloads directory. In the second section, we have an option `app_dir`, which reuses the definition of the downloads directory from the section `paths`.

To realise that, we tell the `ConfigParser` to use the `ExtendedInterpolation` as interpolation type when we instantiate the parser:

```
# interpolation.py
from configparser import ConfigParser
from configparser import ExtendedInterpolation

config = ConfigParser(interpolation=ExtendedInterpolation())
config.read("interpolation_config.ini")
```

> **Note:** An actual interpolation instance needs to be passed to the `ConfigParser`, so do not forget the parentheses `()`.

If we now print the value for the `app_dir` option of the `destinations` section, we get an interpolated string.

```
# Previous code in interpolation.py
print(config.get("destinations", "app_dir"))
```

```
$ python interpolation.py
```

```
/home/florian/Downloads/application/python
```

Summary

Congratulations, you have made it through the article! You not only learnt how to access values from files using the INI-structure, but how to manipulate and extend them, as well. Furthermore, you learnt about the *configparser*'s interpolation capabilities and how to utilise them for your needs.

I hope you enjoyed reading the article. Feel free to share it with your friends and colleagues! Do you have feedback? I am eager to hear it! You can contact me via the contact form or other resources listed in the contact section.

If you have not already, consider following me on Twitter, where I am @DahlitzF, or subscribing to my newsletter! Stay curious and keep coding!

References

1. `ConfigParser` section ↩
2. path-like object glossary entry ↩
3. file object glossary entry ↩

Tweet

Get notified about new articles and more.

Your email Subscribe