ConfigUpdater Documentation

Release 3.0.1

Florian Wilhelm

Jan 21, 2022

CONTENTS

The sole purpose of ConfigUpdater is to easily update an INI config file with no changes to the original file except the intended ones. This means comments, the ordering of sections and key/value-pairs as wells as their cases are kept as in the original file. Thus ConfigUpdater provides complementary functionality to Python's ConfigParser which is primarily meant for reading config files and writing *new* ones. Read more on how to use ConfigUpdater in the *usage page*.

The key differences to ConfigParser are:

- minimal invasive changes in the update configuration file,
- proper handling of comments,
- only a single config file can be updated at a time,
- the original case of sections and keys are kept,
- control over the position of a new section/key

Following features are **deliberately not** implemented:

- interpolation of values,
- propagation of parameters from the default section,
- conversions of values,
- passing key/value-pairs with `default` argument,
- non-strict mode allowing duplicate sections and keys.

Note: ConfigUpdater is mainly developed for PyScaffold.

CHAPTER

ONE

CONTENTS

1.1 Usage

First install the package with:

```
pip install configupdater
```

Now we can simply do:

```
from configupdater import ConfigUpdater

updater = ConfigUpdater()
updater.read("setup.cfg")
```

which would read the file `setup.cfg` that is found in many projects.

To change the value of an existing key we can simply do:

```
updater["metadata"]["author"].value = "Alan Turing"
```

At any point we can print the current state of the configuration file with:

```
print(updater)
```

To update the read-in file just call `updater.update_file()` or `updater.write("filename")` to write the changed configuration file to another destination. Before actually writing, ConfigUpdater will automatically check that the updated configuration file is still valid by parsing it with the help of ConfigParser.

Many of ConfigParser's methods still exists and it's best to look them up in the *module reference*. Let's look at some examples.

1.1.1 Adding and removing options

Let's say we have the following configuration in a string:

```
cfg = """
[metadata]
author = Ada Lovelace
summary = The Analytical Engine
"""
```

We can add an *license* option, i.e. a key/value pair, in the same way we would do with ConfigParser:

```
updater = ConfigUpdater()
updater.read_string(cfg)
updater["metadata"]["license"] = "MIT"
```

A simple `print(updater)` will give show you that the new option was appended to the end:

```
[metadata]
author = Ada Lovelace
summary = The Analytical Engine
license = MIT
```

Since the license is really important to us let's say we want to add it before the `summary` and even add a short comment before it:

```
updater = ConfigUpdater()
updater.read_string(cfg)
(updater["metadata"]["summary"].add_before
                               .comment("Ada would have loved MIT")
                               .option("license", "MIT"))
```

which would result in:

```
[metadata]
author = Ada Lovelace
# Ada would have loved MIT
license = MIT
summary = Analytical Engine calculating the Bernoulli numbers
```

Using `add_after` would give the same result and looks like:

```
updater = ConfigUpdater()
updater.read_string(cfg)
(updater["metadata"]["author"].add_after
                              .comment("Ada would have loved MIT")
                              .option("license", "MIT"))
```

Let's say we want to rename `summary` to the more common `description`:

```
updater = ConfigUpdater()
updater.read_string(cfg)
updater["metadata"]["summary"].key = "description"
```

If we wanted no summary at all, we could just do `del updater["metadata"]["summary"]`.

1.1.2 Adding and removing sections

Adding and remove sections just works like adding and removing options but on a higher level. Sticking to our *Ada Lovelace* example, let's say we want to add a section `options` just before `metadata` with a comment and two new lines to separate it from `metadata`:

```
updater = ConfigUpdater()
updater.read_string(cfg)
(updater["metadata"].add_before
```

(continues on next page)

(continued from previous page)

```
                    .comment("Some specific project options")
                    .section("options")
                    .space(2))
```

As expected, this results in:

```
# Some specific project options
[options]

[metadata]
author = Ada Lovelace
summary = The Analytical Engine
```

We could now fill the new section with options like we learnt before. If we wanted to rename an existing section we could do this with the help of the `name` attribute:

```
updater["metadata"].name = "MetaData"
```

Sometimes it might be useful to inject a new section not in a programmatic way but more declarative. Let's assume we have thus defined our new section in a multi-line string:

```
sphinx_sect_str = """
[build_sphinx]
source_dir = docs
build_dir = docs/_build
"""
```

With the help of two ConfigUpdater objects we can easily inject this section into our example:

```
sphinx = ConfigUpdater()
sphinx.read_string(sphinx_sect_str)
sphinx_sect = sphinx["build_sphinx"]

updater = ConfigUpdater()
updater.read_string(cfg)

(updater["metadata"].add_after
                    .space()
                    .section(sphinx_sect.detach()))
```

The *detach()* method will remove the `build_sphinx` section from the first object and add it to the second object. This results in:

```
[metadata]
author = Ada Lovelace
summary = The Analytical Engine

[build_sphinx]
source_dir = docs
build_dir = docs/_build
```

Alternatively, if you want to preserve `build_sphinx` in both `ConfigUpdater` objects (i.e., prevent it from being removed from the first while still adding a copy to the second), you call also rely on stdlib's `copy.deepcopy()` function instead of *detach()*:

```
from copy import deepcopy

(updater["metadata"].add_after
                    .space()
                    .section(deepcopy(sphinx_sect)))
```

This technique can be used for all objects inside ConfigUpdater: sections, options, comments and blank spaces.

Shallow copies are discouraged in the context of ConfigUpdater because each configuration block keeps a reference to its container to allow easy document editing. When doing editions (such as adding or changing options and comments) based on a shallow copy, the results can be unreliable and unexpected.

For more examples on how the API of ConfigUpdater works it's best to take a look into the unit tests and read the references.

1.2 Contributing

ConfigUpdater is an open-source project and needs your help to improve. If you experience bugs or in general issues, please file an issue report on our issue tracker. If you also want to contribute code or improve the documentation it's best to create a Pull Request (PR) on Github. Here is a short introduction how it works.

1.2.1 Code Contributions

Submit an issue

Before you work on any non-trivial code contribution it's best to first create an issue report to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding we recommend to install Miniconda which allows to setup a dedicated development environment named `configupdater` with:

```
conda create -n configupdater python=3 virtualenv pytest pytest-cov
```

Then activate the environment `configupdater` with:

```
source activate configupdater
```

Clone the repository

1. Create a Gitub account if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on the GitHub server.
3. Clone this copy to your local disk:

   ```
   git clone git@github.com:YourLogin/configupdater.git
   ```

4. Run `python setup.py develop` to install `configupdater` into your environment.

5. Install `pre-commit`:

```
pip install pre-commit
pre-commit install
```

   PyScaffold project comes with a lot of hooks configured to automatically help the developer to check the code being written.

6. Create a branch to hold your changes:

```
git checkout -b my-feature
```

   and start making changes. Never work on the master branch!

7. Start your work on this branch. When you're done editing, do:

```
git add modified_files
git commit
```

   to record your changes in Git, then push them to GitHub with:

```
git push -u origin my-feature
```

8. Please check that your changes don't break any unit tests with:

```
python setup.py test
```

   Don't forget to also add unit tests in case your contribution adds an additional feature and is not just a bugfix.

9. Use flake8 to check your code style.
10. Add yourself to the list of contributors in `AUTHORS.rst`.
11. Go to the web page of your ConfigUpdater fork, and click "Create pull request" to send your changes to the maintainers for review. Find more detailed information creating a PR.

1.3 License

ConfigUpdater is licensed under the MIT license; see below for details.

ConfigUpdater includes code derived from the Python standard library, which is licensed under the Python license, a permissive open source license. The copyright and license are included at the bottom of this file for compliance with Python's terms.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.3.1 A. HISTORY OF THE SOFTWARE

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

> **Release Derived Year Owner GPL-** from compatible? (1)
>
> 0.9.0 thru 1.2 1991-1995 CWI yes 1.3 thru 1.5.2 1.2 1995-1999 CNRI yes 1.6 1.5.2 2000 CNRI no 2.0 1.6 2000 BeOpen.com no 1.6.1 1.6 2001 CNRI yes (2) 2.1 2.0+1.6.1 2001 PSF no 2.0.1 2.0+1.6.1 2001 PSF yes 2.1.1 2.1+2.0.1 2001 PSF yes 2.1.2 2.1.1 2002 PSF yes 2.1.3 2.1.2 2002 PSF yes 2.2 and above 2.1.1 2001-now PSF yes

Footnotes:

(1) GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is "not incompatible" with the GPL.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

1.3.2 B. TERMS AND CONDITIONS FOR ACCESSING OR OTHERWISE USING PYTHON

PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using this software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001, 2002, 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020 Python Software Foundation; All Rights Reserved" are retained in Python alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python.

4. PSF is making Python available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of

agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright (c) 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

1.4 Contributors

- Florian Wilhelm <florian.wilhelm@gmail.com>
- Christian Theune <ct@flyingcircus.io>
- Anderson Bravalheri <andersonbravalheri@gmail.com>

1.5 Changelog

1.5.1 Version 3.0.1

- Fix error when parsing unindented comments in multi-line values, issue #73
- Fix invalid string produced when *allow_no_value = False*, issue #68

1.5.2 Version 3.0

- Added type hinting, issue #16
- Fix parsing error of indented comment lines, issue #25
- Allow handling of raw section comment, issue #25
- More unit testing of optionxform, issue #55
- Allowing sections/options to be copied from one document to the other, issue #47
- New logo, issue #29
- Whole API was rechecked by @abravalheri and changed for consistency, issue #19

1.5.3 Version 2.0

- Changes in parser, i.e. comments in multi-line option values are kept
- Issue #14 is fixed
- Parameter `empty_lines_in_values` is now activated by default and can be changed
- Renamed `sections_blocks` to `section_blocks` for consistency
- Renamed `last_item` to `last_block` for consistency
- Added `first_block`
- Reworked some internal parts of the inheritance hierarchy
- Added `remove` to remove the current block
- Added `next_block` and `previous_block` for easier navigation in section

1.5.4 Version 1.1.3

- Added fallback option to ConfigUpdater.get reflecting ConfigParser

1.5.5 Version 1.1.2

- Fix wrongly modifying input in Option.set_value #11

1.5.6 Version 1.1.1

- Fix iterating over the items() view of a section breaks #8

1.5.7 Version 1.1

- Validate format on write by default (can be deactivated)
- Fixed issue #7 with mixed-case options
- Fixed issue #7 with add_before/add_after problem
- Fixed issue #7 with wrong duplicate mixed-case entries
- Fixed issue #7 with duplicate options after add_after/before

1.5.8 Version 1.0.1

- More sane error message if `read_file` is misused
- Also run unittests with Windows

1.5.9 Version 1.0

- Fix: Use n instead of os.linesep where appropriate

1.5.10 Version 0.3.2

- Added Github and documentation link into the project's metadata

1.5.11 Version 0.3.1

- Require Python >= 3.4 with `python_requires`

1.5.12 Version 0.3

- Added a `insert_at` function at section level
- Some internal code simplifications

1.5.13 Version 0.2

- Added a `to_dict()` function

1.5.14 Version 0.1.1

- Allow for flexible comment character in `comment(...)`

1.5.15 Version 0.1

- First release

1.6 configupdater

1.6.1 configupdater package

Submodules

configupdater.block module

Together with *container* this module forms the basis of the class hierarchy in **ConfigUpdater**.

The *Block* is the parent class of everything that can be nested inside a configuration file, e.g. comments, sections, options and even sequences of white space.

exception configupdater.block.**AlreadyAttachedError**(*block: Union[str, configupdater.block.Block] = 'The block'*)

Bases: `Exception`

{block} has been already attached to a container. Try to remove it first using `detach` or create a copy using stdlib's `copy.deepcopy`.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class configupdater.block.**Block**(*container: Optional[Container] = None*)
Bases: abc.ABC

Abstract Block type holding lines

Block objects hold original lines from the configuration file and hold a reference to a container wherein the object resides.

The type variable T is a reference for the type of the sibling blocks inside the container.

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_line(*line: str*) → configupdater.block.B
PRIVATE: this function is not part of the public API of Block. It is only used internally by other classes of the package during parsing.

Add a line to the current block

Parameters **line** (*str*) – one line to add

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

detach() → configupdater.block.B
Remove and return this block from container

has_container() → bool
Checks if this block has a container attached

property lines: list[str]

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property updated: bool
True if the option was changed/updated, otherwise False

class configupdater.block.**Comment**(*container: Optional[Container] = None*)
Bases: *configupdater.block.Block*

Comment block

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_line(*line: str*) → configupdater.block.B
PRIVATE: this function is not part of the public API of Block. It is only used internally by other classes of the package during parsing.

Add a line to the current block

Parameters **line** (*str*) – one line to add

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

detach() → configupdater.block.B
Remove and return this block from container

has_container() → bool
Checks if this block has a container attached

property lines: list[str]

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property updated: bool
True if the option was changed/updated, otherwise False

exception configupdater.block.**NotAttachedError**(*block: Union[str,* configupdater.block.Block*] = 'The block'*)

Bases: Exception

{block} is not attached to a container yet. Try to insert it first.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class configupdater.block.**Space**(*container: Optional[Container] = None*)
Bases: *configupdater.block.Block*

Vertical space block of new lines

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_line(*line: str*) → configupdater.block.B
PRIVATE: this function is not part of the public API of Block. It is only used internally by other classes of the package during parsing.

Add a line to the current block

Parameters **line** (*str*) – one line to add

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

detach() → configupdater.block.B
Remove and return this block from container

has_container() → bool
Checks if this block has a container attached

property lines: list[str]

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property updated: bool
True if the option was changed/updated, otherwise False

configupdater.builder module

Core of the fluent API used by **ConfigUpdater** to make editing configuration files easier.

class configupdater.builder.**BlockBuilder**(*container: Container*, *idx: int*)
Bases: `object`

Builder that injects blocks at a given index position.

comment(*text: str*, *comment_prefix='#'*) → configupdater.builder.T
Creates a comment block

Parameters

- **text** (*str*) – content of comment without #
- **comment_prefix** (*str*) – character indicating start of comment

Returns self for chaining

option(*key*, *value: Optional[str] = None*, ***kwargs*) → configupdater.builder.T
Creates a new option inside a section

Parameters

- **key** (*str*) – key of the option
- **value** (*str or None*) – value of the option
- ****kwargs** – are passed to the constructor of `Option`

Returns self for chaining

section(*section: Union[str, Section]*) → configupdater.builder.T
Creates a section block

Parameters **section** (str or Section) – name of section or object

Returns self for chaining

space(*newlines: int = 1*) → configupdater.builder.T
Creates a vertical space of newlines

Parameters **newlines** (*int*) – number of empty lines

Returns self for chaining

configupdater.configupdater module

As the main entry point of the ConfigUpdater library, this module is responsible for combining the data layer provided by the *configupdater.document* module and the parsing capabilities of *configupdater.parser*.

To complete the API, this module adds file handling functions, so that you can read a configuration file from the disk, change it to your liking and save the updated content.

exception configupdater.configupdater.**AlreadyAttachedError**(*block: Union[str, configupdater.block.Block] = 'The block'*)

Bases: Exception

{block} has been already attached to a container. Try to remove it first using detach or create a copy using stdlib's copy.deepcopy.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class configupdater.configupdater.**Comment**(*container: Optional[Container] = None*)
Bases: *configupdater.block.Block*

Comment block

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_line(*line: str*) → configupdater.block.B
PRIVATE: this function is not part of the public API of Block. It is only used internally by other classes of the package during parsing.

Add a line to the current block

Parameters **line** (*str*) – one line to add

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

detach() → configupdater.block.B
Remove and return this block from container

has_container() → bool
Checks if this block has a container attached

property lines: list[str]

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property updated: bool
True if the option was changed/updated, otherwise False

class configupdater.configupdater.**ConfigUpdater**(*allow_no_value=False*, ***, *delimiters: Tuple[str, ...] = ('=', ':')*, *comment_prefixes: Tuple[str, ...] = ('#', ';')*, *inline_comment_prefixes: Optional[Tuple[str, ...]] = None*, *strict: bool = True*, *empty_lines_in_values: bool = True*, *space_around_delimiters: bool = True*)

Bases: *configupdater.document.Document*

Tool to parse and modify existing `cfg` files.

ConfigUpdater follows the API of ConfigParser with some differences:

- inline comments are treated as part of a key's value,
- only a single config file can be updated at a time,
- the original case of sections and keys are kept,
- control over the position of a new section/key.

Following features are **deliberately not** implemented:

- interpolation of values,
- propagation of parameters from the default section,
- conversions of values,
- passing key/value-pairs with `default` argument,
- non-strict mode allowing duplicate sections and keys.

ConfigUpdater objects can be created by passing the same kind of arguments accepted by the *Parser*. After a ConfigUpdater object is created, you can load some content into it by calling any of the `read*` methods (*read()*, *read_file()* and *read_string()*).

Once the content is loaded you can use the ConfigUpdater object more or less in the same way you would use a nested dictionary. Please have a look into `Document` to understand the main differences.

When you are done changing the configuration file, you can call *write()* or *update_file()* methods.

add_section(*section: Union[str,* configupdater.section.Section*]*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT.

Parameters **section** (str or *Section*) – name or Section type

append(*block: configupdater.container.T*) → configupdater.container.C

clear() → None. Remove all items from D.

property first_block: Optional[configupdater.container.T]

get(*section*, *option*, *fallback=UniqueValues._UNSET*)
Gets an option value for a given section.

> **Warning:** Please notice this method works differently from what is expected of `MutableMapping.get()` (or `dict.get()`). Similarly to `configparser.ConfigParser.get()`, will take least 2 arguments, and the second argument does not correspond to a default value.
>
> This happens because this function is not designed to return a *Section* of the *ConfigUpdater* document, but instead a nested *Option*.
>
> See *get_section()*, if instead, you want to retrieve a *Section*.

Parameters

- **section** (*str*) – section name
- **option** (*str*) – option name
- **fallback** – if the key is not found and fallback is provided, it will be returned. `None` is a valid fallback value.

Raises

- ***NoSectionError*** – if `section` cannot be found
- ***NoOptionError*** – if the option cannot be found and no `fallback` was given

Returns Option object holding key/value pair

Return type *Option*

get_section(*name*, *default=None*)
This method works similarly to `dict.get()`, and allows you to retrieve an entire section by its name, or provide a `default` value in case it cannot be found.

has_option(*section: str*, *option: str*) → bool
Checks for the existence of a given option in a given section.

Parameters

- **section** (*str*) – name of section
- **option** (*str*) – name of option

Returns whether the option exists in the given section

Return type bool

has_section(*key*) → bool
Returns whether the given section exists.

Parameters **key** (*str*) – name of section

Returns wether the section exists

Return type bool

items(*section=UniqueValues._UNSET*)
Return a list of (name, value) tuples for options or sections.

If section is given, return a list of tuples with (name, value) for each option in the section. Otherwise, return a list of tuples with (section_name, section_type) for each section.

Parameters **section** (*str*) – optional section name, default UNSET

Returns list of *Section* or *Option* objects

Return type list

iter_blocks() → collections.abc.Iterator[configupdater.container.T]
Iterate over all blocks inside container.

iter_sections() → collections.abc.Iterator[*configupdater.section.Section*]
Iterate only over section blocks

keys() → a set-like object providing a view on D's keys

property last_block: Optional[configupdater.container.T]

options(*section: str*) → list[str]
Returns list of configuration options for the named section.

Parameters **section** (*str*) – name of section

Returns list of option names

Return type list

optionxform(*optionstr*) → str
Converts an option key for unification

By default it uses `str.lower()`, which means that ConfigUpdater will compare options in a case insensitive way.

This implementation mimics ConfigParser API, and can be configured as described in `configparser.ConfigParser.optionxform()`.

Parameters **optionstr** (*str*) – key name

Returns unified option name

Return type str

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

read(*filename: str*, *encoding: Optional[str] = None*) → configupdater.configupdater.T
Read and parse a filename.

Parameters

- **filename** (*str*) – path to file
- **encoding** (*str*) – encoding of file, default None

read_file(*f: collections.abc.Iterable[str]*, *source: Optional[str] = None*) → configupdater.configupdater.T
Like read() but the argument must be a file-like object.

The `f` argument must be iterable, returning one line at a time. Optional second argument is the `source` specifying the name of the file being read. If not given, it is taken from f.name. If `f` has no `name` attribute, <???> is used.

Parameters

- **f** – file like object
- **source** (*str*) – reference name for file object, default None

read_string(*string: str*, *source='<string>'*) → configupdater.configupdater.T
Read configuration from a given string.

Parameters

- **string** (*str*) – string containing a configuration
- **source** (*str*) – reference name for file object, default '<string>'

remove_option(*section: str*, *option: str*) → bool
Remove an option.

Parameters

- **section** (*str*) – section name
- **option** (*str*) – option name

Returns whether the option was actually removed

Return type bool

remove_section(*name: str*) → bool
Remove a file section.

Parameters **name** – name of the section

Returns whether the section was actually removed

Return type bool

section_blocks() → list[*configupdater.section.Section*]
Returns all section blocks

Returns list of *Section* blocks

Return type list

sections() → list[str]
Return a list of section names

Returns list of section names

Return type list

set(*section: str*, *option: str*, *value: Union[None, str, collections.abc.Iterable[str]] = None*) →
configupdater.document.D
Set an option.

Parameters

- **section** – section name
- **option** – option name
- **value** – value, default None

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

property structure: list[~T]

property syntax_options: collections.abc.Mapping

to_dict() → dict[str, dict[str, typing.Optional[str]]]
Transform to dictionary

Returns dictionary with same content

Return type dict

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

update_file(*validate: bool = True*) → configupdater.configupdater.T
Update the read-in configuration file.

Parameters **validate** (*Boolean*) – validate format before writing

validate_format(***kwargs*)
Given the current state of the `ConfigUpdater` object (e.g. after modifications), validate its INI/CFG textual representation by parsing it with `ConfigParser`.

The ConfigParser object is instead with the same arguments as the original ConfigUpdater object, but the `kwargs` can be used to overwrite them.

See *validate_format()*.

values() → an object providing a view on D's values

write(*fp: TextIO*, *validate: bool = True*)
Write an .cfg/.ini-format representation of the configuration state.

Parameters

- **fp** (*file-like object*) – open file handle
- **validate** (*Boolean*) – validate format before writing

exception configupdater.configupdater.**NoConfigFileReadError**
Bases: `configparser.Error`

Raised when no configuration file was read but update requested.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception configupdater.configupdater.**NoneValueDisallowed**
Bases: `SyntaxWarning`

Cannot represent <{option} = None>, it will be converted to <{option} = ''>. Please use `allow_no_value=True` with `ConfigUpdater`.

args

classmethod warn(*option*)

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception configupdater.configupdater.**NotAttachedError**(*block: Union[str,* configupdater.block.Block*] = 'The block'*)
Bases: `Exception`

{block} is not attached to a container yet. Try to insert it first.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class configupdater.configupdater.**Option**(*key: str*, *value: Optional[str] = None*, *container: Optional[Section] = None*, *delimiter: str = '='*, *space_around_delimiters: bool = True*, *line: Optional[str] = None*)

Bases: *configupdater.block.Block*

Option block holding a key/value pair.

key
name of the key

Type str

value
stored value

Type str

updated
indicates name change or a new section

Type bool

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_line(*line: str*)
PRIVATE: this function is not part of the public API of Option. It is only used internally by other classes of the package during parsing.

add_value(*value: Optional[str]*)
PRIVATE: this function is not part of the public API of Option. It is only used internally by other classes of the package during parsing.

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

detach() → configupdater.block.B
Remove and return this block from container

has_container() → bool
Checks if this block has a container attached

property key: str
Key string associated with the option.

Please notice that the option key is normalized with *optionxform()*.

When the option object is *detached*, this method will raise a *NotAttachedError*.

property lines: list[str]

property next_block: Optional[configupdater.block.Block]
Next block in the current container

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property raw_key: str
Equivalent to *key*, but before applying `optionxform()`.

property section: *Section*

set_values(*values: collections.abc.Iterable[str]*, *separator='\n'*, *indent=' '*)
Sets the value to a given list of options, e.g. multi-line values

Parameters

- **values** (*iterable*) – sequence of values
- **separator** (*str*) – separator for values, default: line separator
- **indent** (*str*) – indentation depth in case of line separator

property updated: bool
True if the option was changed/updated, otherwise False

property value: Optional[str]

class configupdater.configupdater.**Parser**(*allow_no_value=False, *, delimiters: Tuple[str, ...] = ('=', ':'), comment_prefixes: Tuple[str, ...] = ('#', ';'), inline_comment_prefixes: Optional[Tuple[str, ...]] = None, strict: bool = True, empty_lines_in_values: bool = True, space_around_delimiters: bool = True, optionxform: Callable[[str], str] = <class 'str'>*)

Bases: `object`

Parser for updating configuration files.

ConfigUpdater's parser follows ConfigParser with some differences:

- inline comments are treated as part of a key's value,
- only a single config file can be updated at a time,
- the original case of sections and keys are kept,
- control over the position of a new section/key.

Following features are **deliberately not** implemented:

- interpolation of values,
- propagation of parameters from the default section,
- conversions of values,
- passing key/value-pairs with `default` argument,
- non-strict mode allowing duplicate sections and keys.

NONSPACECRE = re.compile('\\S')

OPTCRE = re.compile('\n (?P<option>.*?) # very permissive!\n \\s*(?P<vi>=|:)\\s* # any number of space/tab,\n # followed by any of t, re.VERBOSE)

OPTCRE_NV = re.compile('\n (?P<option>.*?) # very permissive!\n \\s*(?: # any number of space/tab,\n (?P<vi>=|:)\\s* # optionally followed , re.VERBOSE)

SECTCRE = re.compile('\n \\[# [\n (?P<header>[^]]+) # very permissive!\n \\] #]\n (?P<raw_comment>.*) , re.VERBOSE)

optionxform(*string: str*) → str

read(*filename: str*, *encoding: Optional[str] = None*) → *configupdater.document.Document*
read(*filename: str*, *encoding: str*, *into: configupdater.parser.D*) → configupdater.parser.D
read(*filename: str*, ***, *into: configupdater.parser.D*, *encoding: Optional[str] = 'None'*) → configupdater.parser.D
Read and parse a filename.

Parameters

- **filename** (*str*) – path to file
- **encoding** (*Optional[str]*) – encoding of file, default None
- **into** (*Optional[Document]*) – object to be populated with the parsed config

read_file(*f: collections.abc.Iterable[str]*, *source: Optional[str]*) → *configupdater.document.Document*
read_file(*f: collections.abc.Iterable[str]*, *source: Optional[str]*, *into: configupdater.parser.D*) → configupdater.parser.D
read_file(*f: collections.abc.Iterable[str]*, ***, *into: configupdater.parser.D*, *source: Optional[str] = 'None'*) → configupdater.parser.D
Like read() but the argument must be a file-like object.

The `f` argument must be iterable, returning one line at a time. Optional second argument is the `source` specifying the name of the file being read. If not given, it is taken from f.name. If `f` has no `name` attribute, <???> is used.

Parameters

- **f** – file like object
- **source** (*Optional[str]*) – reference name for file object, default None
- **into** (*Optional[Document]*) – object to be populated with the parsed config

read_string(*string: str*, *source: str = '<string>'*) → *configupdater.document.Document*
read_string(*string: str*, *source: str*, *into: configupdater.parser.D*) → configupdater.parser.D
read_string(*string: str*, ***, *into: configupdater.parser.D*, *source: str = '"<string>"'*) → configupdater.parser.D
Read configuration from a given string.

Parameters

- **string** (*str*) – string containing a configuration
- **source** (*str*) – reference name for file object, default '<string>'
- **into** (*Optional[Document]*) – object to be populated with the parsed config

property syntax_options: collections.abc.Mapping

class configupdater.configupdater.**Section**(*name: str*, *container: Optional[Document] = None*)
Bases: *configupdater.block.Block*, *configupdater.container.Container*[Union[*Option*, *Comment*, *Space*]], collections.abc.MutableMapping[str, Option]

Section block holding options

name
name of the section

Type str

updated
indicates name change or a new section

Type bool

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_comment(*line: str*) → configupdater.section.S
Add a Comment object to the section

Used during initial parsing mainly

Parameters **line** (*str*) – one line in the comment

add_line(*line: str*) → configupdater.block.B
PRIVATE: this function is not part of the public API of Block. It is only used internally by other classes of the package during parsing.

Add a line to the current block

Parameters **line** (*str*) – one line to add

add_option(*entry:* configupdater.option.Option) → configupdater.section.S
Add an Option object to the section

Used during initial parsing mainly

Parameters **entry** (Option) – key value pair as Option object

add_space(*line: str*) → configupdater.section.S
Add a Space object to the section

Used during initial parsing mainly

Parameters **line** (*str*) – one line that defines the space, maybe whitespaces

append(*block: configupdater.container.T*) → configupdater.container.C

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

clear() → None. Remove all items from D.

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

create_option(*key: str*, *value: Optional[str] = None*) → *configupdater.option.Option*
Creates an option with kwargs that respect syntax options given to the parent ConfigUpdater object (e.g. `space_around_delimiters`).

> **Warning:** This is a low level API, not intended for public use. Prefer *set()* or `__setitem__()`.

detach() → configupdater.block.B
Remove and return this block from container

property document: Document

property first_block: Optional[configupdater.container.T]

get(*key: str*) → Optional[*configupdater.option.Option*]
get(*key: str*, *default: configupdater.section.T*) → Union[*configupdater.option.Option*, configupdater.section.T]
This method works similarly to `dict.get()`, and allows you to retrieve an option object by its key.

has_container() → bool
Checks if this block has a container attached

has_option(*key*) → bool
Returns whether the given option exists.

Parameters **option** (*str*) – name of option

Returns whether the section exists

Return type bool

insert_at(*idx: int*) → *configupdater.builder.BlockBuilder*
Returns a builder inserting a new block at the given index

Parameters **idx** (*int*) – index where to insert

items() → list[typing.Tuple[str, *configupdater.option.Option*]]
Return a list of (name, option) tuples for each option in this section.

Returns list of (name, *Option*) tuples

Return type list

iter_blocks() → collections.abc.Iterator[configupdater.container.T]
Iterate over all blocks inside container.

iter_options() → collections.abc.Iterator[*Option*]
Iterate only over option blocks

keys() → a set-like object providing a view on D's keys

property last_block: Optional[configupdater.container.T]

property lines: list[str]

property name: str

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

option_blocks() → list['Option']
Returns option blocks

Returns list of *Option* blocks

Return type list

options() → list[str]
Returns option names

Returns list of option names as strings

Return type list

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

property previous_block: Optional[configupdater.block.Block]
Previous block in the current container

property raw_comment
Raw comment (includes comment mark) inline with the section header

set(*option: str*, *value: Union[None, str, collections.abc.Iterable[str]] = None*) → configupdater.section.S
Set an option for chaining.

Parameters

- **option** – option name
- **value** – value, default None

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

property structure: list[~T]

to_dict() → dict[str, typing.Optional[str]]
Transform to dictionary

Returns dictionary with same content

Return type dict

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

property updated: bool
True if the option was changed/updated, otherwise False

values() → an object providing a view on D's values

class configupdater.configupdater.Space(*container: Optional[Container] = None*)
Bases: configupdater.block.Block

Vertical space block of new lines

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_line(*line: str*) → configupdater.block.B
PRIVATE: this function is not part of the public API of Block. It is only used internally by other classes of the package during parsing.

Add a line to the current block

Parameters **line** (*str*) – one line to add

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

detach() → configupdater.block.B
Remove and return this block from container

has_container() → bool
Checks if this block has a container attached

property lines: list[str]

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property updated: bool
True if the option was changed/updated, otherwise False

configupdater.container module

Together with *block* this module forms the basis of the class hierarchy in **ConfigUpdater**.

The *Container* is the parent class of everything that can contain configuration blocks, e.g. a section or the entire file itself.

class configupdater.container.**Container**
Bases: abc.ABC, Generic[configupdater.container.T]

Abstract Mixin Class describing a container that holds blocks of type T

append(*block: configupdater.container.T*) → configupdater.container.C

property first_block: Optional[configupdater.container.T]

iter_blocks() → collections.abc.Iterator[configupdater.container.T]
Iterate over all blocks inside container.

property last_block: Optional[configupdater.container.T]

property structure: list[~T]

configupdater.document module

This module focus in the top level data layer API of ConfigUpdater, i.e. how to access and modify the sections of the configurations.

Differently from `configparser`, the different aspects of the ConfigUpdater API are split between several modules.

class configupdater.document.**Document**
Bases: *configupdater.container.Container*[Union[Section, Comment, Space]], collections.abc.MutableMapping[str, *configupdater.section.Section*]

Access to the data manipulation API from **ConfigUpdater**.

A `Document` object tries to implement a familiar *dict-like* interface, via `MutableMapping`. However, it also tries to be as compatible as possible with the stdlib's `ConfigParser`. This means that there are a few methods that will work differently from what users familiar with *dict-like* interfaces would expect. The most notable example is *get()*.

A important difference between ConfigUpdater's `Document` model and `ConfigParser` is the behaviour of the `Section` objects. If we represent the type of a *dict-like* (or `MutableMapping`) object by `M[K, V]`, where `K` is the type of its keys and `V` is the type of the associated values, ConfigUpdater's sections would be equivalent to `M[str, Option]`, while ConfigParser's would be `M[str, str]`.

This means that when you try to access a key inside a section in ConfigUpdater, you are going to receive a `Option` object, not its value. To access the value of the option you need to call `Option.value`.

add_section(*section: Union[str,* configupdater.section.Section*]*)
Create a new section in the configuration.

Raise DuplicateSectionError if a section by the specified name already exists. Raise ValueError if name is DEFAULT.

Parameters **section** (str or `Section`) – name or Section type

append(*block: configupdater.container.T*) → configupdater.container.C

clear() → None. Remove all items from D.

property first_block: Optional[configupdater.container.T]

get(*section: str*, *option: str*) → *configupdater.option.Option*
get(*section: str*, *option: str*, *fallback: configupdater.document.T*) → Union[*configupdater.option.Option*, configupdater.document.T]
Gets an option value for a given section.

> **Warning:** Please notice this method works differently from what is expected of `MutableMapping.get()` (or `dict.get()`). Similarly to `configparser.ConfigParser.get()`, will take least 2 arguments, and the second argument does not correspond to a default value.
>
> This happens because this function is not designed to return a `Section` of the `ConfigUpdater` document, but instead a nested `Option`.
>
> See *get_section()*, if instead, you want to retrieve a `Section`.

Parameters

- **section** (*str*) – section name
- **option** (*str*) – option name
- **fallback** – if the key is not found and fallback is provided, it will be returned. `None` is a valid fallback value.

Raises

- ***NoSectionError*** – if `section` cannot be found
- ***NoOptionError*** – if the option cannot be found and no `fallback` was given

Returns Option object holding key/value pair

Return type `Option`

get_section(*name: str*) → Optional[*configupdater.section.Section*]
get_section(*name: str*, *default: configupdater.document.T*) → Union[*configupdater.section.Section*, configupdater.document.T]
This method works similarly to `dict.get()`, and allows you to retrieve an entire section by its name, or provide a `default` value in case it cannot be found.

has_option(*section: str*, *option: str*) → bool
Checks for the existence of a given option in a given section.

Parameters

- **section** (*str*) – name of section

- **option** (*str*) – name of option

Returns whether the option exists in the given section

Return type bool

has_section(*key*) → bool
Returns whether the given section exists.

Parameters **key** (*str*) – name of section

Returns wether the section exists

Return type bool

items() → list[typing.Tuple[str, *configupdater.section.Section*]]
items(*section: str*) → list[typing.Tuple[str, *configupdater.option.Option*]]
Return a list of (name, value) tuples for options or sections.

If section is given, return a list of tuples with (name, value) for each option in the section. Otherwise, return a list of tuples with (section_name, section_type) for each section.

Parameters **section** (*str*) – optional section name, default UNSET

Returns list of `Section` or `Option` objects

Return type list

iter_blocks() → collections.abc.Iterator[configupdater.container.T]
Iterate over all blocks inside container.

iter_sections() → collections.abc.Iterator[*configupdater.section.Section*]
Iterate only over section blocks

keys() → a set-like object providing a view on D's keys

property last_block: Optional[configupdater.container.T]

options(*section: str*) → list[str]
Returns list of configuration options for the named section.

Parameters **section** (*str*) – name of section

Returns list of option names

Return type list

optionxform(*optionstr*) → str
Converts an option key for unification

By default it uses `str.lower()`, which means that ConfigUpdater will compare options in a case insensitive way.

This implementation mimics ConfigParser API, and can be configured as described in `configparser.ConfigParser.optionxform()`.

Parameters **optionstr** (*str*) – key name

Returns unified option name

Return type str

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

remove_option(*section: str*, *option: str*) → bool
Remove an option.

Parameters

- **section** (*str*) – section name
- **option** (*str*) – option name

Returns whether the option was actually removed

Return type bool

remove_section(*name: str*) → bool
Remove a file section.

Parameters **name** – name of the section

Returns whether the section was actually removed

Return type bool

section_blocks() → list[*configupdater.section.Section*]
Returns all section blocks

Returns list of `Section` blocks

Return type list

sections() → list[str]
Return a list of section names

Returns list of section names

Return type list

set(*section: str*, *option: str*, *value: Union[None, str, collections.abc.Iterable[str]] = None*) →
configupdater.document.D
Set an option.

Parameters

- **section** – section name
- **option** – option name
- **value** – value, default None

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

property structure: list[~T]

to_dict() → dict[str, dict[str, typing.Optional[str]]]
Transform to dictionary

Returns dictionary with same content

Return type dict

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

validate_format(***kwargs*)
Call ConfigParser to validate config

Parameters **kwargs** – are passed to `configparser.ConfigParser`

Raises *configparser.ParsingError* – if syntax errors are found

Returns `True` when no error is found

values() → an object providing a view on D's values

configupdater.option module

Options are the ultimate mean of configuration inside a configuration value.

They are always associated with a *key* (or the name of the configuration parameter) and a *value*.

Options can also have multi-line values that are usually interpreted as a list of values.

When editing configuration files with ConfigUpdater, a handy way of setting a multi-line (or comma separated value) for an specific option is to use the *set_values()* method.

exception configupdater.option.**NoneValueDisallowed**

Bases: `SyntaxWarning`

Cannot represent <{option} = None>, it will be converted to <{option} = ''>. Please use `allow_no_value=True` with `ConfigUpdater`.

args

classmethod warn(*option*)

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class configupdater.option.**Option**(*key: str*, *value: Optional[str] = None*, *container: Optional[Section] = None*, *delimiter: str = '='*, *space_around_delimiters: bool = True*, *line: Optional[str] = None*)

Bases: *configupdater.block.Block*

Option block holding a key/value pair.

key

name of the key

Type str

value

stored value

Type str

updated

indicates name change or a new section

Type bool

property add_after: BlockBuilder

Block builder inserting a new block after the current block

property add_before: BlockBuilder

Block builder inserting a new block before the current block

add_line(*line: str*)

PRIVATE: this function is not part of the public API of Option. It is only used internally by other classes of the package during parsing.

add_value(*value: Optional[str]*)
PRIVATE: this function is not part of the public API of Option. It is only used internally by other classes of the package during parsing.

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

detach() → configupdater.block.B
Remove and return this block from container

has_container() → bool
Checks if this block has a container attached

property key: str
Key string associated with the option.

Please notice that the option key is normalized with *optionxform()*.

When the option object is *detached*, this method will raise a `NotAttachedError`.

property lines: list[str]

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property raw_key: str
Equivalent to *key*, but before applying `optionxform()`.

property section: Section

set_values(*values: collections.abc.Iterable[str]*, *separator='\n'*, *indent=' '*)
Sets the value to a given list of options, e.g. multi-line values

Parameters

- **values** (*iterable*) – sequence of values
- **separator** (*str*) – separator for values, default: line separator
- **indent** (*str*) – indentation depth in case of line separator

property updated: bool
True if the option was changed/updated, otherwise False

property value: Optional[str]

configupdater.parser module

Parser for configuration files (normally `*.cfg/*.ini`)

A configuration file consists of sections, lead by a "[section]" header, and followed by "name: value" entries, with continuations and such in the style of RFC 822.

The basic idea of **ConfigUpdater** is that a configuration file consists of three kinds of building blocks: sections, comments and spaces for separation. A section itself consists of three kinds of blocks: options, comments and spaces. This gives us the corresponding data structures to describe a configuration file.

A general block object contains the lines which were parsed and make up the block. If a block object was not changed then during writing the same lines that were parsed will be used to express the block. In case a block, e.g. an option, was changed, it is marked as *updated* and its values will be transformed into a corresponding string during an update of a configuration file.

Note: ConfigUpdater is based on Python's ConfigParser source code, specially regarding the `parser` module. The main parsing rules and algorithm are preserved, however ConfigUpdater implements its own modified version of the abstract syntax tree to support retaining comments and whitespace in an attempt to provide format-preserving document manipulation. The copyright and license of the original ConfigParser code is included as an attachment to ConfigUpdater's own license, at the root of the source code repository; see the file LICENSE for details.

---

exception configupdater.parser.**DuplicateOptionError**(*section*, *option*, *source=None*, *lineno=None*)

Bases: `configparser.Error`

Raised by strict parsers when an option is repeated in an input source.

Current implementation raises this exception only when an option is found more than once in a single file, string or dictionary.

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception configupdater.parser.**DuplicateSectionError**(*section*, *source=None*, *lineno=None*)

Bases: `configparser.Error`

Raised when a section is repeated in an input source.

Possible repetitions that raise this exception are: multiple creation using the API or in strict parsers when a section is found more than once in a single input file, string or dictionary.

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception configupdater.parser.**InconsistentStateError**(*msg*, *fpname='<???>'*, *lineno: int = - 1*, *line: str = '???'*)

Bases: `Exception`

Internal parser error, some of the parsing algorithm assumptions was violated, and the internal state machine ended up in an unpredicted state.

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception configupdater.parser.**MissingSectionHeaderError**(*filename*, *lineno*, *line*)
Bases: *configparser.ParsingError*

Raised when a key-value pair is found before any section header.

append(*lineno*, *line*)

args

property filename
Deprecated, use `source'.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception configupdater.parser.**NoOptionError**(*option*, *section*)
Bases: configparser.Error

A requested option was not found.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception configupdater.parser.**NoSectionError**(*section*)
Bases: configparser.Error

Raised when no section matches a requested option.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class configupdater.parser.**Parser**(*allow_no_value=False, *, delimiters: Tuple[str, ...] = ('=', ':'), comment_prefixes: Tuple[str, ...] = ('#', ';'), inline_comment_prefixes: Optional[Tuple[str, ...]] = None, strict: bool = True, empty_lines_in_values: bool = True, space_around_delimiters: bool = True, optionxform: Callable[[str], str] = <class 'str'>*)
Bases: object

Parser for updating configuration files.

ConfigUpdater's parser follows ConfigParser with some differences:

- inline comments are treated as part of a key's value,
- only a single config file can be updated at a time,
- the original case of sections and keys are kept,
- control over the position of a new section/key.

Following features are **deliberately not** implemented:

- interpolation of values,
- propagation of parameters from the default section,
- conversions of values,
- passing key/value-pairs with default argument,
- non-strict mode allowing duplicate sections and keys.

NONSPACECRE = re.compile('\\S')

OPTCRE = re.compile('\n (?P<option>.*?) # very permissive!\n \\s*(?P<vi>=|:)\\s* # any number of space/tab,\n # followed by any of t, re.VERBOSE)

OPTCRE_NV = re.compile('\n (?P<option>.*?) # very permissive!\n \\s*(?: # any number of space/tab,\n (?P<vi>=|:)\\s* # optionally followed , re.VERBOSE)

SECTCRE = re.compile('\n \\[# [\n (?P<header>[^]]+) # very permissive!\n \\] #]\n (?P<raw_comment>.*) , re.VERBOSE)

optionxform(*string: str*) → str

read(*filename: str*, *encoding: Optional[str] = None*) → *configupdater.document.Document*
read(*filename: str*, *encoding: str*, *into: configupdater.parser.D*) → configupdater.parser.D
read(*filename: str*, *, *into: configupdater.parser.D*, *encoding: Optional[str] = 'None'*) → configupdater.parser.D

Read and parse a filename.

Parameters

- **filename** (*str*) – path to file
- **encoding** (*Optional[str]*) – encoding of file, default None
- **into** (*Optional[Document]*) – object to be populated with the parsed config

read_file(*f: collections.abc.Iterable[str]*, *source: Optional[str]*) → *configupdater.document.Document*
read_file(*f: collections.abc.Iterable[str]*, *source: Optional[str]*, *into: configupdater.parser.D*) → configupdater.parser.D
read_file(*f: collections.abc.Iterable[str]*, *, *into: configupdater.parser.D*, *source: Optional[str] = 'None'*) → configupdater.parser.D

Like read() but the argument must be a file-like object.

The `f` argument must be iterable, returning one line at a time. Optional second argument is the `source` specifying the name of the file being read. If not given, it is taken from f.name. If `f` has no `name` attribute, <???> is used.

Parameters

- **f** – file like object
- **source** (*Optional[str]*) – reference name for file object, default None
- **into** (*Optional[Document]*) – object to be populated with the parsed config

read_string(*string: str*, *source: str = '<string>'*) → *configupdater.document.Document*
read_string(*string: str*, *source: str*, *into: configupdater.parser.D*) → configupdater.parser.D
read_string(*string: str*, *, *into: configupdater.parser.D*, *source: str = '"<string>"'*) → configupdater.parser.D

Read configuration from a given string.

Parameters

- **string** (*str*) – string containing a configuration
- **source** (*str*) – reference name for file object, default '<string>'
- **into** (*Optional[Document]*) – object to be populated with the parsed config

property syntax_options: collections.abc.Mapping

exception configupdater.parser.**ParsingError**(*source=None*, *filename=None*)

Bases: configparser.Error

Raised when a configuration file does not follow legal syntax.

append(*lineno*, *line*)

args

property filename
Deprecated, use `source'.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

configupdater.section module

Sections are intermediate containers in **ConfigUpdater**'s data model for configuration files.

They are at the same time `containers` that hold `options` and `blocks` nested inside the top level configuration *Document*.

Note: Please remember that *Section.get()* method is implemented to mirror the `ConfigParser API` and do not correspond to the more usual `get()` method of *dict-like* objects.

---

class configupdater.section.**Section**(*name: str*, *container: Optional[Document] = None*)
Bases: *configupdater.block.Block*, *configupdater.container.Container*[Union[Option, Comment, Space]], collections.abc.MutableMapping[str, Option]

Section block holding options

name
name of the section

Type str

updated
indicates name change or a new section

Type bool

property add_after: BlockBuilder
Block builder inserting a new block after the current block

property add_before: BlockBuilder
Block builder inserting a new block before the current block

add_comment(*line: str*) → configupdater.section.S
Add a Comment object to the section

Used during initial parsing mainly

Parameters **line** (*str*) – one line in the comment

add_line(*line: str*) → configupdater.block.B
PRIVATE: this function is not part of the public API of Block. It is only used internally by other classes of the package during parsing.

Add a line to the current block

Parameters **line** (*str*) – one line to add

add_option(*entry:* configupdater.option.Option) → configupdater.section.S
Add an Option object to the section

Used during initial parsing mainly

Parameters **entry** (Option) – key value pair as Option object

add_space(*line: str*) → configupdater.section.S
Add a Space object to the section

Used during initial parsing mainly

Parameters **line** (*str*) – one line that defines the space, maybe whitespaces

append(*block: configupdater.container.T*) → configupdater.container.C

attach(*container: Container*) → configupdater.block.B
PRIVATE: Don't use this as a user!

Rather use *add_** or the bracket notation

clear() → None. Remove all items from D.

property container: Container
Container holding the block

property container_idx: int
Index of the block within its container

create_option(*key: str*, *value: Optional[str] = None*) → *configupdater.option.Option*
Creates an option with kwargs that respect syntax options given to the parent ConfigUpdater object (e.g. `space_around_delimiters`).

> **Warning:** This is a low level API, not intended for public use. Prefer `set()` or `__setitem__()`.

detach() → configupdater.block.B
Remove and return this block from container

property document: Document

property first_block: Optional[configupdater.container.T]

get(*key: str*) → Optional[*configupdater.option.Option*]
get(*key: str*, *default: configupdater.section.T*) → Union[*configupdater.option.Option*, configupdater.section.T]
This method works similarly to `dict.get()`, and allows you to retrieve an option object by its key.

has_container() → bool
Checks if this block has a container attached

has_option(*key*) → bool
Returns whether the given option exists.

Parameters **option** (*str*) – name of option

Returns whether the section exists

Return type bool

insert_at(*idx: int*) → *configupdater.builder.BlockBuilder*
Returns a builder inserting a new block at the given index

Parameters **idx** (*int*) – index where to insert

items() → list[typing.Tuple[str, *configupdater.option.Option*]]
Return a list of (name, option) tuples for each option in this section.

Returns list of (name, `Option`) tuples

Return type list

iter_blocks() → collections.abc.Iterator[configupdater.container.T]
Iterate over all blocks inside container.

iter_options() → collections.abc.Iterator[Option]
Iterate only over option blocks

keys() → a set-like object providing a view on D's keys

property last_block: Optional[configupdater.container.T]

property lines: list[str]

property name: str

property next_block: Optional[*configupdater.block.Block*]
Next block in the current container

option_blocks() → list['Option']
Returns option blocks

Returns list of *Option* blocks

Return type list

options() → list[str]
Returns option names

Returns list of option names as strings

Return type list

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

property previous_block: Optional[*configupdater.block.Block*]
Previous block in the current container

property raw_comment
Raw comment (includes comment mark) inline with the section header

set(*option: str*, *value: Union[None, str, collections.abc.Iterable[str]] = None*) → configupdater.section.S
Set an option for chaining.

Parameters

- **option** – option name
- **value** – value, default None

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

property structure: list[~T]

to_dict() → dict[str, typing.Optional[str]]
Transform to dictionary

Returns dictionary with same content

Return type dict

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

property updated: bool
True if the option was changed/updated, otherwise False

values() → an object providing a view on D's values

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

A

B

C

D

F

G

H

I

K

L

M

N

O

T

U

V

W