JSON-Configparser Documentation

Release 0.2.0

Guilherme Ilunga

Jan 25, 2020

Contents

JSON-Configparser is a Python package which enables the usage of JSON files as configuration files which are validated. The main goals of the package are avoiding having to write validation code several times and enabling the usage of JSON files as configuration files.

The package allows parsing several different datatypes from a JSON file:

- integers
- floats
- strings
- booleans
- lists
- dictionaries (with string keys)
- lists/dictionaries of the other types

Besides validating datatypes, it also validates bounds and extra user-defined constraints.

For understanding how to use the package, please check the *Get Started* and *Examples* pages. The API is in the *API* page.

The package is available on PyPI (currently supports Python 3.6 only). To install the latest version, run the following command: `pip install json-configparser`

CHAPTER 1

Get Started

The JSON-Configparser package reads information from a NamedTuple class, which enumerates all arguments, their types, and possibly their defaults too. Besides this the user can define bounds for each argument or other extra validations.

As a result, when reading a JSON file, the package returns a dictionary mapping argument name to value. Then, this dictionary can be used to create an instance of the NamedTuple with the argument values. This instance is immutable (since it is a tuple) and if the user has an IDE, then autocompletion and type checking work when using the object.

The following sections describe the steps of using the package.

1.1 Defining the Arguments

First, the arguments must be defined using the NamedTuple class:

```
from typing import NamedTuple

class Arguments(NamedTuple):
    arg1: int
    arg2: str
    arg3: List[float]
    arg4: Dict[str, bool]
    arg5: bool = True
```

The *Arguments* class defines all arguments and is also a typed NamedTuple. Each argument must have a type. Valid types are:

- `bool`
- `int`
- `float`
- `str`
- `List[x]` where x is any other valid type, including lists and dictionaries

- `Dict[str, x]` where x is any other valid type, including lists and dictionaries

There are certain exceptions during validation: 10.0 is an accepted value for integer arguments and 10 is accepted for float arguments.

Besides types, defaults can also be defined (arg5 in the example is given a default value of True). Every argument defined in the NamedTuple must be given a value in the JSON file, unless it is given a default value. Default values are also type checked by the library. If a default value is given and a value is also found in the JSON, then the value in the JSON is used.

If the JSON contains unknown arguments, i.e., arguments not defined in the NamedTuple, then a ValueError is raised.

1.2 Defining the Bounds

After defining the arguments class, the user can define bounds. Bounds can be defined for arguments of type int, float, or Lists/Dicts of ints or floats.

```
from json_configparser import Bounds

bounds = [Bounds("arg1", lower_bound=1),
          Bounds("arg3", upper_bound=2, upper_inclusive=True)]
```

Bound objects are created by defining the name of the argument (must match the class attribute name), the lower_bound and/or upper_bound, and flags (lower_inclusive and upper_inclusive) indicating if the bound is inclusive or exclusive (default is False, i.e., exclusive).

1.3 Defining the Extra Validations

Finally, besides checking types and defaults, additional user defined validations can be done. The user can define a function which receives a dictionary object, mapping argument name to value. Then, this function can perform checks and raise Errors if an argument is invalid.

```
def extra_validations(args):
    if all(args["arg4"].values()):
        raise ValueError("Not all values in the arg4 argument can be true!")
```

In this case arg4 is a dictionary of booleans which cannot be all True. When parsing the JSON, if all True values are given, the ValueError will be thrown by the library.

This function can also be used to change the values and return a dictionary. However the new values are never checked, so in general this approach is not recommended. Directly changing the args dictionary in this function does not change the final arguments. To do so, the dictionary must be returned by the function.

1.4 Parsing the JSON

The last step is to actually parse a JSON file. To do this, you must first create a *ConfigArgs* object and pass it the *Arguments* class, the bounds list, and the extra validations function defined earlier. You can now directly invoke the *parse_json* method of the *ConfigArgs* instance and pass the path to a valid JSON. If all goes well, this should return a dictionary mapping argument names to values.

Finally, you can instantiate the *Arguments* NamedTuple directly by passing the dictionary. Below is the recommended way of doing these steps.

```
from json_configparser import ConfigArgs

def create_args_object(path_to_json: str):
    args_object = ConfigArgs(Arguments, bounds, extra_validations)
    dict_args = args_object.parse_json(path_to_json)
    return Arguments(**dict_args)
```

For further help, please see the Examples section, or open an issue on Github.

CHAPTER 2

Examples

You can find the code for examples in the examples folder.

2.1 Simple Example

In this example we want to read a list of words, filter them based on their size, and then translate them using a dictionary. Below is a definition of the JSON file (args.json). The *fail* flag indicates if a ValueError should be raised on a word fails to pass the length check.

```
{
    "words": ["hello", "world", "I", "am", "Python", "notproperlytokenizedlongword"],
    "max_size": 10,
    "fail": false,
    "translation": {
        "hello": "Ola",
        "world": "mundo",
        "I": "eu",
        "am": "sou",
        "Python": "Python",
        "notproperlytokenizedlongword": "notproperlytokenizedlongword"
    }
}
```

We must now create a options.py file with the Options NamedTuple definition, the bounds, extra validations, and a function which returns an instance of the NamedTuple, given a path to a valid JSON.

Below is the full options.py file.

```
from typing import NamedTuple, List, Dict
import json_configparser


class Options(NamedTuple):
```

(continues on next page)

(continued from previous page)

```
    words: List[str]
    max_size: int = 2
    fail: bool = False
    translation: Dict[str, str] = {}


bounds = [json_configparser.Bounds("max_size", lower_bound=1)]


def extra_validations(args_dict):
    for word in args_dict["words"]:
        if word not in args_dict["translation"]:
            raise ValueError("Unknown word: {}".format(word))


def create_options_object(path_to_json):
    args_object = json_configparser.ConfigArgs(Options, bounds, extra_validations)
    dict_args = args_object.parse_json(path_to_json)
    return Options(**dict_args)
```

We added some defaults and a lower bound for the max_size argument. Also, we added an extra validation which states that all elements in the words argument should be keys of the translation argument.

Finally, we can implement the main code, which you can find below.

```
import options


def filter_words(args: options.Options):
    filtered_words = []
    for word in args.words:
        if len(word) > args.max_size:
            if args.fail:
                raise ValueError("{} exceeds the maximum size of {} characters".
↪format(word, args.max_size))
            else:
                print("Ignoring '{}', because it exceeds the maximum "
                      "size of {} characters".format(word, args.max_size))
        else:
            filtered_words.append(word)

    return filtered_words


def main(args: options.Options):
    filtered_words = filter_words(args)
    translations = [args.translation[word] for word in filtered_words]
    print(" ".join(translations))


if __name__ == '__main__':
    path_to_json = "args.json"
    main(options.create_options_object(path_to_json))
```

We first instantiate the Options class by calling the options.create_options_object function. From this point, we are now sure that the arguments have been validated and they can no longer be changed. The rest of the code simple uses the attributes of the NamedTuple to filter and translate the words.

CHAPTER 3

API

3.1 bounds module

The Bounds module implements the Bounds class, which can be used to represent bounds for certain arguments.

class json_configparser.bounds.**Bounds**(*arg_name*, *lower_bound=None*, *lower_inclusive=True*, *upper_bound=None*, *upper_inclusive=True*)

Bases: object

Represents the Bounds of an argument. A Bounds instance is defined by lower and upper bounds, and flags indicating if the bounds are inclusive or exclusive. The supported argument types are integer, float, or lists/dictionaries of those types.

__init__(*arg_name*, *lower_bound=None*, *lower_inclusive=True*, *upper_bound=None*, *upper_inclusive=True*)

At least one bound must be provided.

Parameters

- **arg_name** (str) – The name of the argument.
- **lower_bound** (Union[int, float, None]) – The lower bound value.
- **lower_inclusive** (bool) – Flag indicating if the lower bound is inclusive.
- **upper_bound** (Union[int, float, None]) – The upper bound value.
- **upper_inclusive** (bool) – Flag indicating if the lower bound is inclusive.

validate_value(*arg_value*)

Validates a value against the provided bounds.

Parameters **arg_value** (Union[int, float]) – The value of the argument to validate.

Raises **ValueError** – If the value is out of bounds.

3.2 config_args module

Implements the ConfigArgs class which is the main class for parsing options classes, validating arguments, and parsing a JSON file.

class json_configparser.config_args.**ConfigArgs**(*options_class*, *bounds_lst=None*, *extra_validations=None*)

Bases: object

Parses the Arguments NamedTuple class to extract information about argument names, types, and defaults. Also holds information about Bounds and extra validations.

The parse_json method can be used to parse a JSON file and validate it against the known information.

__init__(*options_class*, *bounds_lst=None*, *extra_validations=None*)

Parameters

- **options_class** (type) – The NamedTuple class which defines all arguments, types, and defaults.
- **bounds_lst** (Optional[List[*Bounds*]]) – A list of Bounds objects, which defines bounds for arguments.
- **extra_validations** (Optional[Callable]) – A function which contains extra validations. Should receive a dictionary mapping from argument name to value and should return a dictionary of the same type.

parse_json(*path_to_json*, *encoding='utf-8'*)

Parses a JSON file, reads the arguments, validates them, and returns a dictionary with them.

Parameters

- **path_to_json** (str) – Path to JSON configuration file.
- **encoding** (str) – The encoding to use when loading the JSON file.

Return type Dict[str, Any]

Returns A Dictionary mapping argument name to value.

Raises

- **ValueError** – If an argument is an empty string, if an argument with no default is missing from the JSON, or if the JSON contains an unknown argument.
- **TypeError** – If an argument is of the wrong type.

3.3 type_defaults module

Implements the TypeDefaultBounds NamedTuple, which holds information about an argument.

class json_configparser.type_defaults.**TypeDefaultBounds**

Bases: tuple

NamedTuple representing the name, type, default value, and bounds of an argument.

arg_name

the name of the argument

bound_obj

an instance of the Bound class, representing the bounds of the argument

default_value
the value of the default argument

has_default
flag to indicate if there is a default value for the argument

type_
the type of the argument

3.4 validations module

This module implements the type and bound validation of all supported types.

json_configparser.validations.**validate_argument**(*arg_value*, *arg_type_defaults*)
Given a value and type/bounds, this function checks if the type is supported. If so, then check if the value is of the correct type and if it is within the defined bounds. For Lists and Dictionaries these checks are applied to all elements.

Parameters

- **arg_value** (Any) – The value of the argument to check.
- **arg_type_defaults** (*TypeDefaultBounds*) – An instance of TypeDefaultBounds, specifying the type and bounds of the argument to check.

Raises

- **ValueError** – If the argument is an empty string.
- **TypeError** – If the argument value is not of the expected type.

Return type Any

Python Module Index

j

Index

Symbols

A

B

C

D

H

J

P

T

V