hsd-python

Release 0.1

DFTB+ developers group

Mar 20, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

This package contains utilities to read and write files in the Human-friendly Structured Data (HSD) format.

The HSD-format is very similar to XML, JSON and YAML, but tries to minimize the effort for **humans** to read and write it. It ommits special characters as much as possible (in contrast to XML and JSON) and is not indentation dependent (in contrast to YAML). It was developed originally as the input format for the scientific simulation tool (DFTB+), but is of general purpose. Data stored in HSD can be easily mapped to a subset of JSON, YAML or XML and *vice versa*.

This document describes hsd-python version 0.1.

1.1 Installation

The package can be installed via conda-forge:

```
conda install hsd-python
```

Alternatively, the package can be downloaded and installed via pip into the active Python interpreter (preferably using a virtual python environment) by

```
pip install hsd
```

or into the user space issueing

```
pip install --user hsd
```

1.2 Quick tutorial

A typical, self-explaining input written in HSD looks like

```
driver {
  conjugate_gradients {
    moved_atoms = 1 2 "7:19"
    max_steps = 100
  }
}

hamiltonian {
  dftb {
```

(continues on next page)

(continued from previous page)

```
    scc = yes
    scc_tolerance = 1e-10
    mixer {
      broyden {}
    }
    filling {
      fermi {
        # This is comment which will be ignored
        # Note the attribute (unit) of the field below
        temperature [kelvin] = 100
      }
    }
    k_points_and_weights {
      supercell_folding {
        2   0   0
        0   2   0
        0   0   2
        0.5 0.5 0.5
      }
    }
  }
}
```

The above input can be parsed into a Python dictionary with:

```
import hsd
hsdinput = hsd.load("test.hsd")
```

The dictionary `hsdinput` will then look as:

```
{
    "driver": {
        "conjugate_gradients" {
            "moved_atoms": [1, 2, "7:19"],
            "max_steps": 100
        }
    },
    "hamiltonian": {
        "dftb": {
            "scc": True,
            "scc_tolerance": 1e-10,
            "mixer": {
                "broyden": {}
            },
            "filling": {
                "fermi": {
                    "temperature": 100,
                    "temperature.attrib": "kelvin"
                }
            }
            "k_points_and_weights": {
                "supercell_folding": [
```

(continues on next page)

(continued from previous page)

```
                    [2, 0, 0],
                    [0, 2, 0],
                    [0, 0, 2],
                    [0.5, 0.5, 0.5]
                ]
            }
        }
    }
}
```

Being a simple Python dictionary, it can be easily queried and manipulated in Python

```
hsdinput["driver"]["conjugate_gradients"]["max_steps"] = 200
```

and then stored again in HSD format

```
hsd.dump(hsdinput, "test2.hsd")
```

CHAPTER

TWO

THE HSD FORMAT

2.1 General description

You can think about the Human-readable Structured Data format as a pleasant representation of a tree structure. It can represent a subset of what you can do for example with XML. The following constraints compared to XML apply:

- Every node of a tree, which is not empty, either contains further nodes or data, but never both.
- Every node may have a single (string) attribute only.

These constraints allow a very natural looking formatting of the data.

As an example, let's have a look at a data tree, which represents input for scientific software. In the XML representation, it could be written as

```
<Hamiltonian>
  <Dftb>
    <Scc>Yes</Scc>
    <Filling>
      <Fermi>
        <Temperature attrib="Kelvin">77</Temperature>
      </Fermi>
    <Filling>
  </Dftb>
</Hamiltonian>
```

The same information can be encoded in a much more natural and compact form in HSD format as

```
Hamiltonian {
  Dftb {
    Scc = Yes
    Filling {
      Fermi {
        Temperature [Kelvin] = 77
      }
    }
  }
}
```

The content of a node are passed either between an opening and a closing curly brace or after an equals sign. In the latter case the end of the line will be the closing delimiter. The attribute (typically the unit of the data which the node contains) is specified between square brackets after the node name.

The equals sign can be used to assign data as a node content (provided the data fits into one line), or to assign a single child node as content for a given node. This leads to a compact and expressive notation for those cases, where (by the semantics of the input) a given node is only allowed to have a single child node as content. The tree above is a piece of a typical DFTB+ input, where only one child node is allowed for the nodes `Hamiltonian` and `Filling`, respectively (They specify the type of the Hamiltonian and the filling function). By making use of equals signs, the simplified HSD representation can be as compact as

```
Hamiltonian = Dftb {
  Scc = Yes
  Filling = Fermi {
    Temperature [Kelvin] = 77
  }
}
```

and still represent the same tree.

2.2 Mapping to dictionaries

Being basically a subset of XML, HSD data is best represented as an XML DOM-tree. However, very often a dictionary representation is more desirable, especially when the language used to query and manipulate the tree offers dictionaries as primary data type (e.g. Python). The data in an HSD input can be easily represented with the help of nested dictionaries and lists. The input from the previous section would have the following representation as Python dictionary (or as a JSON formatted input file):

```
{
    "Hamiltonian": {
        "Dftb": {
            "Scc": Yes,
            "Filling": {
                "Fermi": {
                    "Temperature": 77,
                    "Temperature.attrib": "Kelvin"
                }
            }
        }
    }
}
```

The attribute of a node is stored under a special key containting the name of the node and the `.attrib` suffix.

One slight complication of the dictionary representation arises in the case of node which has multiple child nodes with the same name

```
<ExternalField>
  <PointCharges>
    <GaussianBlurWidth>3</GaussianBlurWidth>
    <CoordsAndCharges>
     3.3 -1.2 0.9   9.2
     1.2 -3.4 5.6  -3.3
    </CoordsAndCharges>
  </PointCharges>
  <PointCharges>
    <GaussianBlurWidth>10</GaussianBlurWidth>
```

(continues on next page)

(continued from previous page)

```
    <CoordsAndCharges>
     1.0   2.0  3.0  4.0
     -1.0 -2.0 -3.0 -4.0
    </CoordsAndCharges>
  </PointCharges>
</ExternalField>
```

While the HSD representation has no problem to cope with the situation

```
ExternalField {
  PointCharges {
    GaussianBlurWidth = 3
    CoordsAndCharges {
     3.3 -1.2 0.9   9.2
     1.2 -3.4 5.6  -3.3
    }
  }
  PointCharges {
    GaussianBlurWidth = 10
    CoordsAndCharges {
     1.0   2.0  3.0  4.0
     -1.0 -2.0 -3.0 -4.0
    }
  }
}
```

a trick is needed for the dictionary / JSON representation, as multiple keys with the same name are not allowed in a dictionary. Therefore, the repetitive nodes will be mapped to one key, which will contain a list of dictionaries (instead of a single dictionary as in the usual case):

```
{
    "ExternalField": {
        // Note the list of dictionaries here!
        "PointCharges": [
            {
                "GaussianBlurWidth": 3,
                "CoordsAndCharges": [
                    [3.3, -1.2, 0.9, 9.2],
                    [1.2, -3.4, 5.6, -3.3]
                ]
            },
            {
                "GaussianBlurWidth": 10,
                "CoordsAndCharges": [
                    [1.0,  2.0, 3.0, 4.0 ],
                    [-1.0, -2.0, -3.0, -4.0 ]
                ]
            },
        ]
        # Also attributes becomes a list. Due to technialc reasons the
        # dictbuilder always creates an attribute list for mulitple nodes,
        # even if none of the nodes carries an actual attribute.
```

(continues on next page)

(continued from previous page)

```
        "PointCharges.attrib": [None, None]
    }
}
```

The mapping works in both directions, so that this dictionary (or the JSON file created from it) can be easily converted back to the HSD form again.

2.3 Processing related information

Additional to the data stored in an HSD-file, further processing related information can be recorded on demand. The current Python implementation is able to record following additional data for each HSD node:

- the line, where the node was defined in the input (helpful for printing out informative error messages),
- the name of the HSD node, as found in the input (useful if the tag names are converted to lower case to ease case-insensitive handling of the input) and
- whether an equals sign was used to open the block.

If this information is being recorded, a special key with the `.hsdattrib` suffix will be generated for each node in the dictionary/JSON presentation. The corresponding value will be a dictionary with those information.

As an example, let's store the input from the previous section

```
Hamiltonian = Dftb {
  Scc = Yes
  Filling = Fermi {
    Temperature [Kelvin] = 77
  }
}
```

in the file *test.hsd*, parse it and convert the node names to lower case (to make enable case-insensitive input processing). Using the Python command

```
inpdict = hsd.load("test.hsd", lower_tag_names=True, include_hsd_attribs=True)
```

will yield the following dictionary representation of the input:

```
{
    'hamiltonian.hsdattrib': {'equal': True, 'line': 0, 'tag': 'Hamiltonian'},
    'hamiltonian': {
        'dftb.hsdattrib': {'line': 0, equal: False, 'tag': 'Dftb'},
        'dftb': {
            'scc.hsdattrib': {'equal': True, 'line': 1, 'tag': 'Scc'},
            'scc': True,
            'filling.hsdattrib': {'equal': True, 'line': 2, 'tag': 'Filling'},
            'filling': {
                'fermi.hsdattrib': {'line': 2, 'equal': False, 'tag': 'Fermi'},
                'fermi': {
                    'temperature.attrib': 'Kelvin',
                    'temperature.hsdattrib': {'equal': True, 'line': 3,
                                              'tag': 'Temperature'},
                    'temperature': 77
```

(continues on next page)

(continued from previous page)

```
                }
            }
        }
    }
}
```

The recorded line numbers can be used to issue helpful error messages with information about where the user should search for the problem. The node names and formatting information about the equal sign ensures that the formatting is similar to the original HSD, if the data is dumped into the HSD format again. Dumping the dictionary with

```
hsd.dump(inpdict, "test2-formatted.hsd", use_hsd_attribs=True)
```

would indeed yield

```
Hamiltonian = Dftb {
  Scc = Yes
  Filling = Fermi {
    Temperature [Kelvin] = 77
  }
}
```

which is basically identical with the original input. If the additional processing information is not recorded when the data is loaded, or it is not considered when the data is dumped as HSD again

```
inpdict = hsd.load("test.hsd", lower_tag_names=True)
hsd.dump(inpdict, "test2-unformatted.hsd")
```

the resulting formatting will more strongly differ from the original HSD

```
hamiltonian {
  dftb {
    scc = Yes
    filling {
      fermi {
        temperature [Kelvin] = 77
      }
    }
  }
}
```

Still nice and readable, but less compact and with different casing.

CHAPTER

THREE

API DOCUMENTATION

3.1 High level routines

hsd.**load_string**(*hsdstr: str*, *lower_tag_names: bool = False*, *include_hsd_attribs: bool = False*, *flatten_data: bool = False*) → dict

Loads a string with HSD-formatted data into a Python dictionary.

Parameters

- **hsdstr** – String with HSD-formatted data.
- **lower_tag_names** – When set, all tag names will be converted to lower-case (practical, when input should be treated case insensitive.) If `include_hsd_attribs` is set, the original tag name will be stored among the HSD attributes.
- **include_hsd_attribs** – Whether the HSD-attributes (processing related attributes, like original tag name, line information, etc.) should be stored. Use it, if you wish to keep the formatting of the data close to the original one on writing (e.g. lowered tag names converted back to their original form, equals signs between parent and only child kept, instead of converted to curly braces).
- **flatten_data** – Whether multiline data in the HSD input should be flattened into a single list. Othewise a list of lists is created, with one list for every line (default).

Returns

Dictionary representing the HSD data.

Examples

```
>>> hsdstr = """
... Dftb {
...   Scc = Yes
...   Filling {
...     Fermi {
...       Temperature [Kelvin] = 100
...     }
...   }
... }
... """
>>> hsd.load_string(hsdstr)
{'Dftb': {'Scc': True, 'Filling': {'Fermi': {'Temperature': 100,
'Temperature.attrib': 'Kelvin'}}}}
```

In order to ease the case-insensitive handling of the input, the tag names can be converted to lower case during reading using the `lower_tag_names` option.

```
>>> hsd.load_string(hsdstr, lower_tag_names=True)
{'dftb': {'scc': True, 'filling': {'fermi': {'temperature': 100,
'temperature.attrib': 'Kelvin'}}}}
```

The original tag names (together with additional information like the line number of a tag) can be recorded, if the `include_hsd_attribs` option is set:

```
>>> data = hsd.load_string(hsdstr, lower_tag_names=True,
... include_hsd_attribs=True)
```

Each tag in the dictionary will have a corresponding ".hsdattrib" entry with the recorded data:

```
>>> data["dftb.hsdattrib"]
{'equal': False, 'line': 1, 'name': 'Dftb'}
```

This additional data can be then also used to format the tags in the original style, when writing the data in HSD-format again. Compare:

```
>>> hsd.dump_string(data)
'dftb {\n  scc = Yes\n  filling {\n    fermi {\n
temperature [Kelvin] = 100\n    }\n  }\n}\n'
```

versus

```
>>> hsd.dump_string(data, use_hsd_attribs=True)
'Dftb {\n  Scc = Yes\n  Filling {\n    Fermi {\n
Temperature [Kelvin] = 100\n    }\n  }\n}\n'
```

hsd.load(*hsdfile: Union[TextIO, str]*, *lower_tag_names: bool = False*, *include_hsd_attribs: bool = False*, *flatten_data: bool = False*) → dict

Loads a file with HSD-formatted data into a Python dictionary

Parameters

- **hsdfile** – Name of file or file like object to read the HSD data from
- **lower_tag_names** – When set, all tag names will be converted to lower-case (practical, when input should be treated case insensitive.) If `include_hsd_attribs` is set, the original tag name will be stored among the HSD attributes.
- **include_hsd_attribs** – Whether the HSD-attributes (processing related attributes, like original tag name, line information, etc.) should be stored. Use it, if you wish to keep the formatting of the data close to the original on writing (e.g. lowered tag names converted back to their original form, equals signs between parent and only child kept, instead of converted to curly braces).
- **flatten_data** – Whether multiline data in the HSD input should be flattened into a single list. Othewise a list of lists is created, with one list for every line (default).

Returns

Dictionary representing the HSD data.

Examples

See *hsd.load_string()* for examples of usage.

hsd.**dump_string**(*data: dict*, *use_hsd_attribs: bool = False*) → str

Serializes an object to string in HSD format.

Parameters

- **data** – Dictionary like object to be written in HSD format.
- **use_hsd_attribs** – Whether HSD attributes of the data structure should be used to format the output (e.g. to restore original mixed case tag names)

Returns

HSD formatted string.

Examples

```
>>> hsdtree = {
...     'Dftb': {
...         'Scc': True,
...         'Filling': {
...             'Fermi': {
...                 'Temperature': 100,
...                 'Temperature.attrib': 'Kelvin'
...             }
...         }
...     }
... }
>>> hsd.dump_string(hsdtree)
'Dftb {\n  Scc = Yes\n  Filling {\n    Fermi {\n
Temperature [Kelvin] = 100\n    }\n  }\n}\n'
```

See also *hsd.load_string()* for an example.

hsd.**dump**(*data: dict*, *hsdfile: Union[TextIO, str]*, *use_hsd_attribs: bool = False*)

Dumps data to a file in HSD format.

Parameters

- **data** – Dictionary like object to be written in HSD format
- **hsdfile** – Name of file or file like object to write the result to.
- **use_hsd_attribs** – Whether HSD attributes in the data structure should be used to format the output.

  This option can be used to for example to restore original tag names, if the file was loaded with the `lower_tag_names` and `include_hsd_attribs` options set or keep the equal signs between parent and contained only child.

Raises

TypeError – if object is not a dictionary instance.

Examples

See *hsd.load_string()* for an example.

3.2 Lower level building blocks

class hsd.**HsdParser**(*eventhandler: Optional[HsdEventHandler] = None*)

Event based parser for the HSD format.

Parameters
eventhandler – Object which should handle the HSD-events triggered during parsing. When not specified, HsdEventPrinter() is used.

Examples

```
>>> from io import StringIO
>>> dictbuilder = hsd.HsdDictBuilder()
>>> parser = hsd.HsdParser(eventhandler=dictbuilder)
>>> hsdfile = StringIO("""
... Hamiltonian {
...     Dftb {
...         Scc = Yes
...         Filling = Fermi {
...             Temperature [Kelvin] = 100
...         }
...     }
... }
... """)
>>> parser.parse(hsdfile)
>>> dictbuilder.hsddict
{'Hamiltonian': {'Dftb': {'Scc': True, 'Filling': {'Fermi':
{'Temperature': 100, 'Temperature.attrib': 'Kelvin'}}}}}
```

parse(*fobj: Union[TextIO, str]*)

Parses the provided file-like object.

The parser will process the data and trigger the corresponding events in the eventhandler which was passed at initialization.

Parameters
fobj – File like object or name of a file containing the data.

class hsd.**HsdEventHandler**

Abstract base class for handling HSD events.

abstract open_tag(*tagname: str*, *attrib: Optional[str]*, *hsdattrib: Optional[dict]*)

Opens a tag.

Parameters

- **tagname** – Name of the tag which had been opened.
- **attrib** – String containing the attribute of the tag or None.
- **hsdattrib** – Dictionary of the options created during the processing in the hsd-parser.

abstract close_tag(*tagname: str*)

Closes a tag.

Parameters
tagname – Name of the tag which had been closed.

abstract add_text(*text: str*)

Adds text (data) to the current tag.

Parameters
text – Text in the current tag.

class hsd.HsdDictBuilder(*flatten_data: bool = False*, *lower_tag_names: bool = False*, *include_hsd_attribs: bool = False*)

Specific HSD event handler, which builds a nested Python dictionary.

Parameters

- **flatten_data** – Whether multiline data in the HSD input should be flattened into a single list. Othewise a list of lists is created, with one list for every line (default).
- **lower_tag_names** – Whether tag names should be all converted to lower case (to ease case insensitive processing). Default: False. If set and include_hsd_attribs is also set, the original tag names can be retrieved from the "name" hsd attributes.
- **include_hsd_attribs** – Whether the HSD-attributes (processing related attributes, like original tag name, line information, etc.) should be stored (default: False).

property hsddict

The dictionary which has been built

open_tag(*tagname*, *attrib*, *hsdattrib*)

Opens a tag.

Parameters

- **tagname** – Name of the tag which had been opened.
- **attrib** – String containing the attribute of the tag or None.
- **hsdattrib** – Dictionary of the options created during the processing in the hsd-parser.

close_tag(*tagname*)

Closes a tag.

Parameters
tagname – Name of the tag which had been closed.

add_text(*text*)

Adds text (data) to the current tag.

Parameters
text – Text in the current tag.

class hsd.HsdDictWalker(*eventhandler: Optional[HsdEventHandler] = None*)

Walks through a Python dictionary and triggers HSD events.

Parameters
eventhandler – Event handler dealing with the HSD events generated while walking through the dictionary. When not specified, the events are printed.

walk(*dictobj*)

Walks through the directory and generates HSD events.

Parameters
dictobj – Directory to walk through.

class hsd.**HsdFormatter**(*fobj*, *use_hsd_attribs=True*)

Implements an even driven HSD formatter.

Parameters

- **fobj** – File like object to write the formatted output to.
- **use_hsd_attribs** – Whether HSD attributes passed to the formatter should be considered, when formatting the the output (default: True)

open_tag(*tagname: str*, *attrib: str*, *hsdattrib: dict*)

Opens a tag.

Parameters

- **tagname** – Name of the tag which had been opened.
- **attrib** – String containing the attribute of the tag or None.
- **hsdattrib** – Dictionary of the options created during the processing in the hsd-parser.

close_tag(*tagname: str*)

Closes a tag.

Parameters
tagname – Name of the tag which had been closed.

add_text(*text: str*)

Adds text (data) to the current tag.

Parameters
text – Text in the current tag.

INDEX