PyNMRSTAR

A Python module for reading, writing, and manipulating NMR-STAR files.
build passing

Python versions supported: 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6

Overview

This library was developed by the BMRB to give the Python-using NMR community tools to work with the NMR-STAR data format. It is used internally and is actively maintained. The library is thoroughly documented such that calling `help(object_or_method)` from an interactive python session will print the documentation for the object or method.

That same documentation, as well as some notes on module-level variables is located here.
Finally, there are several command-line based tools developed to enable simple queries to pull data out of an NMR-STAR file. Those tools also serve as great examples of how to use the library. You can view those here.

Introduction to NMR-STAR

To understand how the library works, you first need to understand the NMR-STAR terminology and file format. If you are already familiar with NMR-STAR, feel free to jump ahead to the section on this library.

A NMR-STAR entry/file is componsed of one or more saveframes (conceptually you should think of a saveframe as a data block), each of which contain tags and loops. There can only be one of each tag in a saveframe. If a tag has multiple values, the only way to represent it is to place it inside a loop. A loop is simply a set of tags with multiple values.

Therefore, hierarchically, you can picture a NMR-STAR file as a tree where the entry is the trunk, the large branches are the saveframes, and each saveframe may contain one or more loops - the branches.

Here is a very simple example of a NMR-STAR file:

```
data_dates
    save_special_dates_saveframe_1
        _Special_Dates.Type     Holidays
        loop_
            _Events.Date
            _Events.Desciption
            12/31/2017 "New Year's Eve"
            01/01/2018 "New Year's Day"
        stop_
    save_
```

In the previous example, the entry name is `dates` because that is what follows the `data_` tag. Next, there is one saveframe, with a name of `special_dates_saveframe_1` and a tag prefix (which corresponds to the saveframe category) of `Special_Dates`. There is one tag in the saveframe, with a tag name of `Type` and a value of `Holidays`. There is also one loop of category `events` that has information about two different events (though an unlimited number of events could be present).

The first datum in each row corresponds to the first tag, `Date`, and the second corresponds to the second tag, `Description`.

Values in NMR-STAR format need to be quoted if they contain a space, tab, vertical tab, or newline in the value. This library takes care of that for you, but it is worth knowing. That is why in the example the dates are not quoted, but the event descriptions are.

Quick Start to PyNMRSTAR

First, pull up an interactive python session and import the module:

```
>>> import pynmrstar
```

There are many ways to load an NMR-STAR entry, but lets focus on the most common two.

From the BMRB API (loads the most up to date version of an entry from the BMRB API):

```
>>> entry15000 = pynmrstar.Entry.from_database(15000)
```

From a file:

```
>>> entry = pynmrstar.Entry.from_file("/location/of/the/file.
str")
```

Continuing on we will assume you have loaded entry 15000 from the API using the from_database command.

Writing out a modified entry or saveframe to file is just as easy:

```
>>> entry15000.write_to_file("output_file_name.str")
```

Viewing the structure of the entry

To see the overall structure of the entry, use the `print_tree()` method.

```
>>> entry15000.print_tree()
<pynmrstar.Entry '15000' from_database(15000)>
    [0] <pynmrstar.Saveframe 'entry_information'>
        [0] <pynmrstar.Loop '_Entry_author'>
        [1] <pynmrstar.Loop '_SG_project'>
        [2] <pynmrstar.Loop '_Struct_keywords'>
        [3] <pynmrstar.Loop '_Data_set'>
        [4] <pynmrstar.Loop '_Datum'>
        [5] <pynmrstar.Loop '_Release'>
        [6] <pynmrstar.Loop '_Related_entries'>
    [1] <pynmrstar.Saveframe 'citation_1'>
        [0] <pynmrstar.Loop '_Citation_author'>
    [2] <pynmrstar.Saveframe 'assembly'>
        [0] <pynmrstar.Loop '_Entity_assembly'>
    [3] <pynmrstar.Saveframe 'F5-Phe-cVHP'>
        [0] <pynmrstar.Loop '_Entity_db_link'>
        [1] <pynmrstar.Loop '_Entity_comp_index'>
        [2] <pynmrstar.Loop '_Entity_poly_seq'>
    [4] <pynmrstar.Saveframe 'natural_source'>
        [0] <pynmrstar.Loop '_Entity_natural_src'>
    [5] <pynmrstar.Saveframe 'experimental_source'>
        [0] <pynmrstar.Loop '_Entity_experimental_src'>
    [6] <pynmrstar.Saveframe 'chem_comp_PHF'>
        [0] <pynmrstar.Loop '_Chem_comp_descriptor'>
        [1] <pynmrstar.Loop '_Chem_comp_atom'>
        [2] <pynmrstar.Loop '_Chem_comp_bond'>
    [7] <pynmrstar.Saveframe 'unlabeled_sample'>
        [0] <pynmrstar.Loop '_Sample_component'>
    [8] <pynmrstar.Saveframe 'selectively_labeled_sample'>
        [0] <pynmrstar.Loop '_Sample_component'>
    [9] <pynmrstar.Saveframe 'sample_conditions'>
        [0] <pynmrstar.Loop '_Sample_condition_variable'>
    [10] <pynmrstar.Saveframe 'NMRPipe'>
        [0] <pynmrstar.Loop '_Vendor'>
        [1] <pynmrstar.Loop '_Task'>
    [11] <pynmrstar.Saveframe 'PIPP'>
        [0] <pynmrstar.Loop '_Vendor'>
        [1] <pynmrstar.Loop '_Task'>
    [12] <pynmrstar.Saveframe 'SPARKY'>
        [0] <pynmrstar.Loop '_Vendor'>
        [1] <pynmrstar.Loop '_Task'>
    [13] <pynmrstar.Saveframe 'CYANA'>
        [0] <pynmrstar.Loop '_Vendor'>
        [1] <pynmrstar.Loop '_Task'>
    [14] <pynmrstar.Saveframe 'X-PLOR_NIH'>
        [0] <pynmrstar.Loop '_Vendor'>
        [1] <pynmrstar.Loop '_Task'>
    [15] <pynmrstar.Saveframe 'spectrometer_1'>
    [16] <pynmrstar.Saveframe 'spectrometer_2'>
    [17] <pynmrstar.Saveframe 'spectrometer_3'>
    [18] <pynmrstar.Saveframe 'spectrometer_4'>
    [19] <pynmrstar.Saveframe 'spectrometer_5'>
    [20] <pynmrstar.Saveframe 'spectrometer_6'>
    [21] <pynmrstar.Saveframe 'NMR_spectrometer_list'>
        [0] <pynmrstar.Loop '_NMR_spectrometer_view'>
    [22] <pynmrstar.Saveframe 'experiment_list'>
        [0] <pynmrstar.Loop '_Experiment'>
    [23] <pynmrstar.Saveframe 'chemical_shift_reference_1'>
        [0] <pynmrstar.Loop '_Chem_shift_ref'>
    [24] <pynmrstar.Saveframe 'assigned_chem_shift_list_1'>
        [0] <pynmrstar.Loop '_Chem_shift_experiment'>
        [1] <pynmrstar.Loop '_Atom_chem_shift'>
```

You can see that there are 24 saveframes, and each saveframe contains some number of loops.

Accessing saveframes and loops

There are several ways to access saveframes and loops depending on what you hope to accomplish.

The interactive session way

When playing with the library, debugging, or learning about NMR-STAR you will most likely find the following method most convenient. Note that it is not the correct pattern to use if you want to iterate all of the data in an entry (for reasons that will be explained below).

You can access the saveframes in an entry directly using their *names*. For example, to get a reference to the spectrometer saveframe named `spectrometer_1` you can simply do the following:

```
>>> a_spectrometer = entry15000['spectrometer_1']
```

Note that you can see the saveframe names in the tree printout above.

You can do the same for loops within a saveframe, but for loops you must use their tag category (the part before the period) to access them (note that to get to the `Vendor` loop we first had to go through its parent saveframe, named `X-PLOR_NIH` (the `X-PLOR_NIH` saveframe is of the category `software` - you'll see where you access the category later and why accessing by category is preferrable).

```
>>> explor_nih_vendor = entry15000['X-PLOR_NIH']['_Vendor']
>>> print explor_nih_vendor
```

```
    loop_
       _Vendor.Name
       _Vendor.Address
       _Vendor.Electronic_address
       _Vendor.Entry_ID
       _Vendor.Software_ID


     'CD Schwieters, JJ Kuszewski, N Tjandra and GM Clore'
.   .   15000   5


    stop_

```

These shortcuts are there for your convenience when writing code. The reason you shouldn't use them in production code is because the saveframe names - what you use as a reference - can actually have any arbitrary value. They are fairly consistent, and for certain saveframes are always the same, but for other saveframes users can set them to whatever value they want during the deposition. Therefore the much better way to access data is via the *category*. Note that only one saveframe in an entry can have a given name, but multiple saveframes may be of the same category.

The `_` prior to the `Vendor` loop category is to make it clear you want to access the loop and not a saveframe tag with the name `Vendor`.

The robust (and recommended) way

A better way to access data is via the category of the data you want to read, or by searching for it with a full tag name. Before going into detail, take a look at what one saveframe from the entry above looks like:

```
############################
#  Computer software used  #
############################


save_X-PLOR_NIH
   _Software.Sf_category   software
   _Software.Sf_framecode  X-PLOR_NIH
   _Software.Entry_ID      15000
   _Software.ID            5
   _Software.Name          'X-PLOR NIH'
   _Software.Version       .
   _Software.Details       .


   loop_
      _Vendor.Name
      _Vendor.Address
      _Vendor.Electronic_address
      _Vendor.Entry_ID
      _Vendor.Software_ID


     'CD Schwieters, JJ Kuszewski, N Tjandra and GM Clore'
.   .   15000   5


```

```
   stop_

   loop_
      _Task.Task
      _Task.Entry_ID
      _Task.Software_ID

     refinement             15000   5
     'structure solution'   15000   5

   stop_

save_
```

This is a saveframe describing software that was used during an NMR study. You can see from the saveframe tags that the name of this software package is X-PLOR-NIH. You can see from the tag `ID` that it is the fifth software saveframe in this entry. The category of this saveframe is “software” which you can see in the `Sf_category` (short for saveframe category) tag.

This saveframe also has two loops, a vendor loop and a task loop. These are loops rather than free tags as a given software package can have more than one vendor and more than one task it performs.

Reading the software packages

The more robust way to access the data in the software saveframes is by iterating over all of the software saveframes in the entry and pulling out the data we want. To do this for software, we would write the following:

```
>>> software_saveframes = entry15000.get_saveframes_by_catego
ry('software')
>>> software_saveframes
[<pynmrstar.Saveframe 'NMRPipe'>,
 <pynmrstar.Saveframe 'PIPP'>,
 <pynmrstar.Saveframe 'SPARKY'>,
 <pynmrstar.Saveframe 'CYANA'>,
 <pynmrstar.Saveframe 'X-PLOR_NIH'>]
```

You can see that this method, `get_saveframes_by_category` returned all of the software saveframes in the entry. Now we can iterate through them to either pull out data, modify data, or remove data. (One note, each loop category - the text before the period in the loop tags - is unique to its parent saveframe. Therefore you will never find a `Task` loop in a saveframe with a category of anything other than `software`. Furthermore, a saveframe can only have one loop of a given category. This means that accessing loops within a saveframe using the category notation is robust and will not lead to you missing a loop.)

The following will combine all the task loops in the entry into CSV format.

```
>>> csv_data = ""
>>> for software_sf in software_saveframes:
>>>     print_header = True
>>>    # Wrap this in try/catch because it is not gauranteed a software saveframe will have a task loop
>>>    try:
>>>        csv_data += software_sf['_Task'].get_data_as_csv(header=print_header)
>>>        print_header = False
>>>    except KeyError:
>>>        continue
>>> csv_data
'_Task.Task,_Task.Entry_ID,_Task.Software_ID\nprocessing,15000,1\n_Task.Task,_Task.Entry_ID,_Task.Software_ID\nchemical shift assignment,15000,2\ndata analysis,15000,2\npeak picking,15000,2\n_Task.Task,_Task.Entry_ID,_Task.Software_ID\nchemical shift assignment,15000,3\n_Task.Task,_Task.Entry_ID,_Task.Software_ID\nstructure solution,15000,4\n_Task.Task,_Task.Entry_ID,_Task.Software_ID\nrefinement,15000,5\nstructure solution,15000,5\n'
```

Using get_tag to pull tags directly from an entry

Another way to access data in by using the full tag name. Keep in mind that a full tag contains a category first, a period, and then a tag name. So if we wanted to see all of the various `_Task.Task` that the software packages associated with this entry performed, a simple way to do so is with the `get_tag()` method of the entry:

```
>>> entry15000.get_tag('Task.Task')
```

```
[u'processing',
 u'chemical shift assignment',
 u'data analysis',
 u'peak picking',
 u'chemical shift assignment',
 u'structure solution',
 u'refinement',
 u'structure solution']
```

Or to get all of the spectrometer information - `get_tags()` accepts a list of tags to fetch and returns a dictionary pointing to all the values of each tag, with the order preserved:

```
>>> entry15000.get_tags(['_NMR_spectrometer.Manufacturer', '_NMR_spectrometer.Model', '_NMR_spectrometer.Field_strength'])
{'_NMR_spectrometer.Field_strength': [u'500',
  u'500',
  u'750',
  u'600',
  u'800',
  u'900'],
 '_NMR_spectrometer.Manufacturer': [u'Bruker',
  u'Bruker',
  u'Bruker',
  u'Varian',
  u'Varian',
  u'Varian'],
 '_NMR_spectrometer.Model': [u'Avance',
  u'Avance',
  u'Avance',
  u'INOVA',
  u'INOVA',
  u'INOVA']}
```

To view all of the tags in the NMR-STAR schema and their meanings, please go here.

Assigned Chemical Shifts

“I just want to get the chemical shift data as an array - how do I do that?”

Keep in mind that an entry may have multiple sets of assigned chemical shifts. (For examples, there made be two sets of assignments that were made under two differerent sample conditions.) So to get the chemical shifts it is best to iterate through all the assigned chemical shift loops:

```
>>> cs_result_sets = []
>>> for chemical_shift_loop in entry15000.get_loops_by_catego
ry("Atom_chem_shift"):
>>>     cs_result_sets.append(chemical_shift_loop.get_tag(['C
omp_index_ID', 'Comp_ID', 'Atom_ID', 'Atom_type', 'Val', 'Val
_err']))
>>> cs_result_sets
[[[u'2', u'SER', u'H', u'H', u'9.3070', u'0.01'],
  [u'2', u'SER', u'HA', u'H', u'4.5970', u'0.01'],
  [u'2', u'SER', u'HB2', u'H', u'4.3010', u'0.01'],
  [u'2', u'SER', u'HB3', u'H', u'4.0550', u'0.01'],
  [u'2', u'SER', u'CB', u'C', u'64.6000', u'0.1'],
  [u'2', u'SER', u'N', u'N', u'121.5800', u'0.1'],
  [u'3', u'ASP', u'H', u'H', u'8.0740', u'0.01'],
  [u'3', u'ASP', u'HA', u'H', u'4.5580', u'0.01'],
  [u'3', u'ASP', u'HB2', u'H', u'2.835', u'0.01'],
  ...
```

Note that we used the `get_tag()` method of the loop to only pull out the tags we were concerned with. `get_tag()` accepts an array of tags in addition to a single tag. The full assigned chemical saveframe loop will contain extra tags you may not need. For example:

```
>>> print entry15000.get_loops_by_category("Atom_chem_shift")
[0]
   loop_
      _Atom_chem_shift.ID
      _Atom_chem_shift.Assembly_atom_ID
      _Atom_chem_shift.Entity_assembly_ID
      _Atom_chem_shift.Entity_ID
      _Atom_chem_shift.Comp_index_ID
      _Atom_chem_shift.Seq_ID
      _Atom_chem_shift.Comp_ID
      _Atom_chem_shift.Atom_ID
      _Atom_chem_shift.Atom_type
      _Atom_chem_shift.Atom_isotope_number
      _Atom_chem_shift.Val
      _Atom_chem_shift.Val_err
      _Atom_chem_shift.Assign_fig_of_merit
      _Atom_chem_shift.Ambiguity_code
      _Atom_chem_shift.Occupancy
      _Atom_chem_shift.Resonance_ID
      _Atom_chem_shift.Auth_entity_assembly_ID
      _Atom_chem_shift.Auth_asym_ID
      _Atom_chem_shift.Auth_seq_ID
      _Atom_chem_shift.Auth_comp_ID
      _Atom_chem_shift.Auth_atom_ID
      _Atom_chem_shift.Details
      _Atom_chem_shift.Entry_ID
      _Atom_chem_shift.Assigned_chem_shift_list_ID

     1     .   1   1   2    2    SER   H      H   1    9.3070
     0.01   .   .   .   .   .   .   2    SER   H     .   1500
0   1
     2     .   1   1   2    2    SER   HA     H   1    4.5970
     0.01   .   .   .   .   .   .   2    SER   HA    .   1500
0   1
     3     .   1   1   2    2    SER   HB2    H   1    4.3010
     0.01   .   .   .   .   .   .   2    SER   HB2   .   1500
0   1
    ...
```

“But I want to access the chemical shifts as numbers, not strings!”

That is easy to do. When you first load an entry it is by default loaded with all values as strings. To instead load it such that the values match the schema, simply turn on CONVERT_DATATYPES prior to loading it.

```
>>> pynmrstar.CONVERT_DATATYPES = True
>>> ent15000 = pynmrstar.Entry.from_database(15000)
>>> cs_result_sets = []
>>> for chemical_shift_loop in entry15000.get_loops_by_category("Atom_chem_shift"):
>>>     cs_result_sets.append(chemical_shift_loop.get_tag(['Comp_index_ID', 'Comp_ID', 'Atom_ID', 'Atom_type', 'Val', 'Val_err']))
>>> cs_result_sets
[[[2, u'SER', u'H', u'H', Decimal('9.3070'), Decimal('0.01')],
  [2, u'SER', u'HA', u'H', Decimal('4.5970'), Decimal('0.01')],
  [2, u'SER', u'HB2', u'H', Decimal('4.3010'), Decimal('0.01')],
  [2, u'SER', u'HB3', u'H', Decimal('4.0550'), Decimal('0.01')],
  [2, u'SER', u'CB', u'C', Decimal('64.6000'), Decimal('0.1')],
  [2, u'SER', u'N', u'N', Decimal('121.5800'), Decimal('0.1')],
  [3, u'ASP', u'H', u'H', Decimal('8.0740'), Decimal('0.01')]
,
  [3, u'ASP', u'HA', u'H', Decimal('4.5580'), Decimal('0.01')
],
  [3, u'ASP', u'HB2', u'H', Decimal('2.835'), Decimal('0.01')
],
  [3, u'ASP', u'HB3', u'H', Decimal('2.754'), Decimal('0.01')
],
  [3, u'ASP', u'CA', u'C', Decimal('57.6400'), Decimal('0.1')
],
  [3, u'ASP', u'N', u'N', Decimal('121.1040'), Decimal('0.1')
],
    ...
```

This is a great opportunity to point out that if all you want is the chemical shifts, or one or two tags, you may find it significantly easier to use the BMRB API (chemical shift endpoint) to fetch that data directly and on-demand rather than dealing directly with NMR-STAR at all.

Creating new loops and saveframes

This tutorial has so far focused on how to read and access data. This section will focus on how to create new loop and saveframe objects.

Loops

There are five ways to make a new loop: `from_file()`, `from_json()`, `from_scratch()`, `from_string()`, and `from_template()`. All of these are classmethods. `from_scratch()` makes a new loop, `from_string()` parses an NMR-STAR loop from a python string containing NMR-STAR data, `from_json()` parses a JSON object (reversely, `get_json()` will get a JSON representation of the loop), `from_scratch()` makes a completely empty loop, and `from_template()` makes a loop with the tags prefilled from the BMRB schema based on the provided category. `from_file`, `from_json`, and `from_string` are fairly self-explanatory - see the full documentation if needed for usage.

`from_scratch()`

```
>>> lp = pynmrstar.Loop.from_scratch()
>>> print lp

   loop_

   stop_

>>> lp.add_tag(['loop_category.tag1', 'loop_category.tag2', '
loop_category.tag3'])
>>> print lp

   loop_
      _loop_category.tag1
      _loop_category.tag2
      _loop_category.tag3
   stop_

# Note that when calling add_data the length of the array mus
t match the number of tags in the loop
>>> lp.add_data(['value_1', 2, 'value 3'])
>>> print lp
   loop_
      _loop_category.tag1
      _loop_category.tag2
      _loop_category.tag3

     value_1   2   'value 3'

   stop_

# Alternatively, you can (with caution) directly modify the a
rray corresponding to the loop data
>>> lp.data = [[1,2,3],[4,5,6]]
>>> print lp
   loop_
      _loop_category.tag1
      _loop_category.tag2
      _loop_category.tag3

     1   2   3
     4   5   6
```

```
    stop_
```

Note that the loop category was set automatically when the tag `loop_category.tag1` was added. You could have also provided the tag when creating the loop by providing it as an argument to the optional `category` argument to the constructor.

`from_template()`

This method will create a new loop ready for data with the tags from the BMRB schema corresponding to that loop category.

```
>>> chemical_shifts = pynmrstar.Loop.from_template('atom_chem_shift_list')
>>> print chemical_shifts
    loop_
        _Atom_chem_shift.ID
        _Atom_chem_shift.Assembly_atom_ID
        _Atom_chem_shift.Entity_assembly_ID
        _Atom_chem_shift.Entity_ID
        _Atom_chem_shift.Comp_index_ID
        _Atom_chem_shift.Seq_ID
        _Atom_chem_shift.Comp_ID
        _Atom_chem_shift.Atom_ID
        _Atom_chem_shift.Atom_type
        _Atom_chem_shift.Atom_isotope_number
        _Atom_chem_shift.Val
```

```
      _Atom_chem_shift.Val_err
      _Atom_chem_shift.Assign_fig_of_merit
      _Atom_chem_shift.Ambiguity_code
      _Atom_chem_shift.Ambiguity_set_ID
      _Atom_chem_shift.Occupancy
      _Atom_chem_shift.Resonance_ID
      _Atom_chem_shift.Auth_entity_assembly_ID
      _Atom_chem_shift.Auth_asym_ID
      _Atom_chem_shift.Auth_seq_ID
      _Atom_chem_shift.Auth_comp_ID
      _Atom_chem_shift.Auth_atom_ID
      _Atom_chem_shift.Details
      _Atom_chem_shift.Entry_ID
      _Atom_chem_shift.Assigned_chem_shift_list_ID



   stop_
```

Saveframes

There are five ways to make a new loop: `from_file()`, `from_json()`, `from_scratch()`, `from_string()`, and `from_template()`. All of these are classmethods. `from_scratch()` makes a new saveframe, `from_string()` parses an NMR-STAR saveframe from a python string containing NMR-STAR data, `from_json()` parses a JSON object (reversely, `get_json()` will get a JSON representation of the saveframe), `from_scratch()` makes a completely empty saveframe, and `from_template()` makes a saveframe with the tags prefilled from the BMRB schema based on the provided category. `from_file`, `from_json`, and `from_string` are fairly self-explanatory - see the full documentation if needed for usage.

`from_scratch()`

```
# You must provide the saveframe name (the value that comes after "save_" at the start of the saveframe and saveframe tag prefix (the value before the "." in a tag name) when creating a saveframe this way
>>> my_sf = pynmrstar.Saveframe.from_scratch("sf_name", "example_sf_category")
>>> print my_sf
save_sf_name

save_

# Add a tag using the add_tag() method. Update=True will override existing tag with the same name. Update=False will raise an exception if the tag already exists
>>> my_sf.add_tag("tagName1", "tagValue1")
>>> print my_sf
save_sf_name
   _example_sf_category.tagName1  tagValue1

save_

```

```
>>> my_sf.add_tag("tagName1", "tagValue2", update=False)
ValueError: There is already a tag with the name 'tagName1'.
>>> my_sf.add_tag("tagName1", "tagValue2", update=True)
>>> print my_sf
save_sf_name
   _example_sf_category.tagName1  tagValue1

save_
# Alternatively, you can access or write tag values using dir
ect subset access:
>>> my_sf['tagName1']
['tagValue2']
>>> my_sf['tagName2'] = "some value"
>>> print my_sf
save_sf_name
   _example_sf_category.tagName1  tagValue2
   _example_sf_category.tagName2  'some value'

save_

# Now add the loop we created before
>>> my_sf.add_loop(lp)
>>> print my_sf
save_sf_name
   _example_sf_category.tagName1  tagValue2
   _example_sf_category.tagName2  'some value'

   loop_
```

```
       _loop_category.tag1
       _loop_category.tag2
       _loop_category.tag3

      1   2   3
      4   5   6

    stop_

save_

# Now write out our saveframe to a file. Optionally specify f
ormat="json" to write in JSON format.
>>> my_sf.write_to_file("file_name.str")
>>> my_sf.write_to_file("file_name.json", format_="json")
```

`from_template()`

```

>>> my_sf = pynmrstar.Saveframe.from_template("assigned_chemi
cal_shifts")
>>> print my_sf
print my_sf
     ###################################
     #  Assigned chemical shift lists  #
     ###################################

#############################################################
```

```
######
#       Chemical Shift Ambiguity Index Value Definitions
     #
#
     #
# The values other than 1 are used for those atoms with diffe
rent #
# chemical shifts that cannot be assigned to stereospecific a
toms #
# or to specific residues or chains.
     #
#
     #
#   Index Value            Definition
     #
#
     #
#      1             Unique (including isolated methyl proton
s,   #
#                         geminal atoms, and geminal methyl
     #
#                         groups with identical chemical shif
ts)  #
#                         (e.g. ILE HD11, HD12, HD13 protons)
     #
#      2             Ambiguity of geminal atoms or geminal me
thyl #
#                         proton groups (e.g. ASP HB2 and HB3
     #
#                         protons, LEU CD1 and CD2 carbons, o
r    #
#                         LEU HD11, HD12, HD13 and HD21, HD22
,    #
#                         HD23 methyl protons)
     #
#     3              Aromatic atoms on opposite sides of
     #
#                         symmetrical rings (e.g. TYR HE1 and
 HE2 #
#                         protons)
     #
#     4              Intraresidue ambiguities (e.g. LYS HG an
d    #
#                         HD protons or TRP HZ2 and HZ3 proto
ns)  #
#     5              Interresidue ambiguities (LYS 12 vs. LYS
 27) #
#     6              Intermolecular ambiguities (e.g. ASP 31
CA   #
#                         in monomer 1 and ASP 31 CA in monom
er 2 #
#                         of an asymmetrical homodimer, duple
x    #
#                         DNA assignments, or other assignmen
ts   #
#                         that may apply to atoms in one or m
```

```
ore  #
#                         molecule in the molecular assembly)
     #
#      9             Ambiguous, specific ambiguity not define
d    #
#
     #
#############################################################
######

save_assigned_chemical_shifts
   _Assigned_chem_shift_list.Sf_category                  ass
igned_chemical_shifts
   _Assigned_chem_shift_list.Sf_framecode                 ass
igned_chemical_shifts
   _Assigned_chem_shift_list.Entry_ID                     .
   _Assigned_chem_shift_list.ID                           .
   _Assigned_chem_shift_list.Sample_condition_list_ID     .
   _Assigned_chem_shift_list.Sample_condition_list_label  .
   _Assigned_chem_shift_list.Chem_shift_reference_ID      .
   _Assigned_chem_shift_list.Chem_shift_reference_label   .
   _Assigned_chem_shift_list.Chem_shift_1H_err            .
   _Assigned_chem_shift_list.Chem_shift_13C_err           .
   _Assigned_chem_shift_list.Chem_shift_15N_err           .
   _Assigned_chem_shift_list.Chem_shift_31P_err           .
   _Assigned_chem_shift_list.Chem_shift_2H_err            .
   _Assigned_chem_shift_list.Chem_shift_19F_err           .
   _Assigned_chem_shift_list.Error_derivation_method      .
   _Assigned_chem_shift_list.Details                             .
   _Assigned_chem_shift_list.Text_data_format                    .
   _Assigned_chem_shift_list.Text_data                           .

   loop_
      _Chem_shift_experiment.Experiment_ID
      _Chem_shift_experiment.Experiment_name
      _Chem_shift_experiment.Sample_ID
      _Chem_shift_experiment.Sample_label
      _Chem_shift_experiment.Sample_state
      _Chem_shift_experiment.Entry_ID
      _Chem_shift_experiment.Assigned_chem_shift_list_ID


   stop_

   loop_
      _Systematic_chem_shift_offset.Type
      _Systematic_chem_shift_offset.Atom_type
      _Systematic_chem_shift_offset.Atom_isotope_number
      _Systematic_chem_shift_offset.Val
      _Systematic_chem_shift_offset.Val_err
      _Systematic_chem_shift_offset.Entry_ID
      _Systematic_chem_shift_offset.Assigned_chem_shift_list_
ID


   stop_
```

```


    loop_
       _Chem_shift_software.Software_ID
       _Chem_shift_software.Software_label
       _Chem_shift_software.Method_ID
       _Chem_shift_software.Method_label
       _Chem_shift_software.Entry_ID
       _Chem_shift_software.Assigned_chem_shift_list_ID



    stop_


    loop_
       _Atom_chem_shift.ID
       _Atom_chem_shift.Assembly_atom_ID
       _Atom_chem_shift.Entity_assembly_ID
       _Atom_chem_shift.Entity_ID
       _Atom_chem_shift.Comp_index_ID
       _Atom_chem_shift.Seq_ID
       _Atom_chem_shift.Comp_ID
       _Atom_chem_shift.Atom_ID
       _Atom_chem_shift.Atom_type
       _Atom_chem_shift.Atom_isotope_number
       _Atom_chem_shift.Val
       _Atom_chem_shift.Val_err
       _Atom_chem_shift.Assign_fig_of_merit
       _Atom_chem_shift.Ambiguity_code
       _Atom_chem_shift.Ambiguity_set_ID
```

```
      _Atom_chem_shift.Occupancy
      _Atom_chem_shift.Resonance_ID
      _Atom_chem_shift.Auth_entity_assembly_ID
      _Atom_chem_shift.Auth_asym_ID
      _Atom_chem_shift.Auth_seq_ID
      _Atom_chem_shift.Auth_comp_ID
      _Atom_chem_shift.Auth_atom_ID
      _Atom_chem_shift.Details
      _Atom_chem_shift.Entry_ID
      _Atom_chem_shift.Assigned_chem_shift_list_ID


   stop_

   loop_
      _Ambiguous_atom_chem_shift.Ambiguous_shift_set_ID
      _Ambiguous_atom_chem_shift.Atom_chem_shift_ID
      _Ambiguous_atom_chem_shift.Entry_ID
      _Ambiguous_atom_chem_shift.Assigned_chem_shift_list_ID


   stop_

save_

```

Schema methods

The library makes it easy to add missing tags, sort the tags according to the BMRB schema, and validate the data against the schema. Let’s do a simple example of creating a chemical shift loop, adding any missing tags, ordering the tags in the standard order (not required), and then checking for errors.

```
# Create the loop with the proper category
>>> my_cs_loop = pynmrstar.Loop.from_scratch("Atom_chem_shift")
# Add the tags we will fill
>>> my_cs_loop.add_tag(['Comp_ID', 'Atom_ID', 'Comp_index_ID', 'Atom_type', 'Val', 'Val_err'])
   loop_
      _Atom_chem_shift.Comp_ID
      _Atom_chem_shift.Atom_ID
      _Atom_chem_shift.Comp_Index_ID
      _Atom_chem_shift.Atom_type
      _Atom_chem_shift.Val
      _Atom_chem_shift.Val_err



   stop_
# Populate the data array
>>> my_cs_loop.data = [['SER', 'H',  '2', 'H', '9.3070', '0.01'],
                       ['SER', 'HA', '2', 'H', '4.5970', '0.01'],
                        ['SER', 'HB2', '2', 'H', '4.3010', '0.
01']]
>>> print my_cs_loop
   loop_
      _Atom_chem_shift.Comp_ID
      _Atom_chem_shift.Atom_ID
      _Atom_chem_shift.Comp_Index_ID
      _Atom_chem_shift.Atom_type
      _Atom_chem_shift.Val
      _Atom_chem_shift.Val_err

     SER   H     2   H   9.3070   0.01
     SER   HA    2   H   4.5970   0.01
     SER   HB2   2   H   4.3010   0.01

   stop_

# Now lets sort the tags to match the BMRB schema
>>> my_cs_loop.sort_tags()
# You can see that the Comp_index_ID tag has been moved to th
e front to match the BMRB standard
>>> print my_cs_loop
   loop_
      _Atom_chem_shift.Comp_index_ID
      _Atom_chem_shift.Comp_ID
      _Atom_chem_shift.Atom_ID
      _Atom_chem_shift.Atom_type
      _Atom_chem_shift.Val
```

```
       _Atom_chem_shift.Val_err

      2    SER    H      H    9.3070    0.01
      2    SER    HA     H    4.5970    0.01
      2    SER    HB2    H    4.3010    0.01

   stop_

# Check for any errors - returns a list of errors. No errors
here:
>>> print my_cs_loop.validate()
[]
# Let us now set 'Comp_index_ID' to have an invalid value
>>> my_cs_loop.data[0][0] = "invalid"
# You can see that there is now a validation error - the data
 doesn't match the specified type
>>> print my_cs_loop.validate()
["Value does not match specification: '_Atom_chem_shift.Comp_
index_ID':'invalid' on line '0 tag 0 of loop'.\n     Type spe
cified: int\n     Regular expression for type: '-?[0-9]+'"]
# If you use the pynmrstar.validate(object) function, it will
 print the report in a human-readable format
>>> pynmrstar.validate(my_cs_loop)
1: Value does not match specification: '_Atom_chem_shift.Comp
_index_ID':'invalid' on line '0 tag 0 of loop'.
     Type specified: int
     Regular expression for type: '-?[0-9]+'

```

```
# Finally, add in any tags that you didn't have a value for
>>> my_cs_loop.add_missing_tags()
# You can see that all the standard "Atom_chem_shift" loop ta
gs have been added, and their values all set to a logical nul
l value - "."
>>> print my_cs_loop

   loop_
      _Atom_chem_shift.ID
      _Atom_chem_shift.Assembly_atom_ID
      _Atom_chem_shift.Entity_assembly_ID
      _Atom_chem_shift.Entity_ID
      _Atom_chem_shift.Comp_index_ID
      _Atom_chem_shift.Seq_ID
      _Atom_chem_shift.Comp_ID
      _Atom_chem_shift.Atom_ID
      _Atom_chem_shift.Atom_type
      _Atom_chem_shift.Atom_isotope_number
      _Atom_chem_shift.Val
      _Atom_chem_shift.Val_err
      _Atom_chem_shift.Assign_fig_of_merit
      _Atom_chem_shift.Ambiguity_code
      _Atom_chem_shift.Ambiguity_set_ID
      _Atom_chem_shift.Occupancy
      _Atom_chem_shift.Resonance_ID
      _Atom_chem_shift.Auth_entity_assembly_ID
      _Atom_chem_shift.Auth_asym_ID
      _Atom_chem_shift.Auth_seq_ID
      _Atom_chem_shift.Auth_comp_ID
      _Atom_chem_shift.Auth_atom_ID
      _Atom_chem_shift.Details
      _Atom_chem_shift.Entry_ID
      _Atom_chem_shift.Assigned_chem_shift_list_ID

     .   .   .   .   invalid   .   SER   H     H   .   9.3070
    0.01   .   .   .   .   .   .   .   .   .   .   .   .   .

     .   .   .   .   2         .   SER   HA    H   .   4.5970
    0.01   .   .   .   .   .   .   .   .   .   .   .   .   .

     .   .   .   .   2         .   SER   HB2   H   .   4.3010
    0.01   .   .   .   .   .   .   .   .   .   .   .   .   .


    stop_
```

For more examples of PyNMRSTAR library usage, please look here.
For the full documentation of all available methods and classes, please look here.

For any questions or suggestions, please create an issue on the GitHub page.