# Programming with PyCIFRW and PySTARRW

March 8, 2006

PyCIFRW provides facilities for reading, manipulating and writing CIF and STAR files. In addition, CIF files and dictionaries may be validated against DDL1/2 dictionaries.

## 1 Installing and Initialising PyCIFRW

Assuming python is installed, you can unpack the distribution into a temporary directory, and then type "`python setup.py`" from within this temporary directory. Upon completion of this command, the python distutils installer will have placed four files into the python packages directory: `CifFile.py`, `StarFile.py, yappsrt.py` and `YappsStarParser.py`. It is then sufficient to import `CifFile.py` to access all CIF features:

```
>>> import CifFile
```

## 2 Working with CIF files

### 2.1 Creating a CifFile object

CIF files are represented in PyCIFRW as `CifFile` objects. These objects behave identically to Python dictionaries, with some additional methods. CIF files can be created by calling the `ReadCif` function:

```
>>> cf = CifFile.ReadCif("mycif.cif")
```

Optionally, the maximum input line length can be specified; it defaults to the CIF 1.1 value of 2048 characters.

Errors are raised if CIF syntax/grammar violations are encountered in the input file or line length limits are exceeded.

#### 2.1.1 Creating a new CifFile

A new `CifFile` object is usually created empty:

```
cf = CifFile.CifFile()
```

You will need to create at least one `CifBlock` object to hold your data:

```
myblock = CifFile.CifBlock()
cf['a_block'] = myblock
```

A `CifBlock` object may be initialised with another `CifBlock`, in which case a copy operation is performed, or with a tuple or list of tuples containing key, value pairs. These are inserted into the new `CifBlock` using `AddCifItem` (see below).

## 2.2 Manipulating values in a CIF file

### 2.2.1 Accessing data

The simplest form of access is using standard Python square bracket notation. Data blocks and data names within each data block are referenced identically to normal Python dictionaries:

```
my_data = cf['a_data_block']['_a_data_name']
```

All values are strings with CIF syntactical elements stripped, that is, no enclosing quotation marks or semicolons are included in the values. The value associated with a `CifFile` dictionary key is always a `CifBlock` object. All standard Python dictionary methods (e.g. `get, update, items, keys`) are available for both `CifFile` and `CifBlock` objects.

If a data name occurs in a loop, a list of string values is returned for the value of that dataname. However, in practice, looped data is usually only useful in combination with other values from the same loop. `CifBlock` method `GetLoop(dataname)` will return all data in the loop containing dataname as a `CifLoopBlock` object, which provides the same methods as the `CifBlock`. For example, `keys()` returns a list of datanames in the loop. Additionally, loop packets can be accessed by accessing the nth value in the `CifLoopBlock` object:

```
>>> lb = cb.GetLoop("_item_5")
>>> lb[0]
['1', 'a', '5']
```

which also means that it is simple to loop over all packets using default Python behaviour:

```
>>> for j in lb: print `j`
['1', 'a', '5']
['2', 'b', '6']
['3', 'c', '7']
['4', 'd', '8']
```

The corresponding datanames are accessible through method GetItemOrder:

```
>>> lb.GetItemOrder()
['_item_5', '_item_7', '_item_6']
```

An alternative way of accessing loop data is through the built in `CifLoopBlock` iterator called `flat_iterator`, which on each call to the iterator object returns a `CifLoopBlock` with single-valued datanames (for most purposes identical to a Python dictionary):

```
>>> t = lb.flat_iterator()
>>> for a in t: print `a["_item_7"]`
'a' 'b' 'c' 'd'
```

### 2.2.2 Changing or adding data values

If many operations are going to be performed on a single data block, it is convenient to assign that block to a new variable:

```
cb = cf['my_block']
```

A new data name and value may be added, or the value of an existing name changed, by straight assignment:

```
cb['_new_data_name'] = 4.5
cb['_old_data_name'] = 'cucumber'
```

Old values are overwritten silently. Note that values may be strings or numbers.

If a list is given as the value instead of a single string or number, a new loop is created containing this one data name, looped. If this data name already appeared in a loop, any looped data values which may have co-occurred in the loop are deleted. As this is not necessarily the desired behaviour, you may wish to access the loop block using the `GetLoop` method described above.

Alternatively, the underlying AddCifItem method can be used to add multiple looped and unlooped data items in a single command. `AddCifItem` is called with a 2-element tuple argument. The first element of the tuple is either a single dataname, or a list or tuple of datanames. The second element is either a single value (in the case of a single name in the first element) or a list, each element of which is a list of values taken by the corresponding dataname in the first element. A nested tuple of datanames in the first element together with the corresponding nested tuple of lists in the second element will become a loop block in the Cif file.

Another method, `AddToLoop(dataname,newdata)`, adds `newdata` to the pre-existing loop containing `dataname`, silently overwriting duplicate data. `Newdata` should be a Python dictionary of dataname - datavalue pairs, with `datavalue` a list of new/replacement values.

Note that lists (and objects) returned by PyCIFRW actually represent the list currently inside the CifBlock, and therefore any modification to them will modify the stored list. While this is often the desired behaviour, if you intend to alter any such lists in other parts of your program while maintaining CIF integrity, you should first copy them to avoid destroying the loop structure:

```
mysym = cb['_symmetry_ops'][:]
mysym.append('x-1/2,y+1/2,z')
```

**Examples using loops**

**Adding/replacing a single item with looped values:**

```
cb['_symmetry'] = ['x,y,z','-x,-y,-z','x+1/2,y,z']
```

results in an output fragment

```
loop_
   _symmetry
   x,y,z
   -x,-y,-z
   x+1/2,y,z
```

**Adding a complete loop:**

```
cb.AddCifItem((['_example','_example_detail']],
              [['123.4','4567.8'],
               ['small cell','large cell']]]))
```

results in an output fragment:

```
loop_
   _example
   _example_detail
   123.4 'small cell'
   4567.8 'large cell'
```

**Appending a new dataname to a pre-existing loop:**

```
cb.AddToLoop(
      '_example',{'_comment':["not that small","Big and beautiful"]}
            )
```

changes the previous output to be

```
loop_
   _example
   _example_detail
   _comment
   123.4 'small cell' 'not that small'
   4567.8 'large cell' 'Big and beautiful'
```

**Changing pre-existing data in a loop:**

```
cb.AddToLoop('_comment',{'_example':['12.2','12004']})
```

changes the previous example to

```
loop_
   _example
   _example_detail
   _comment
   12.2 'small cell' 'not that small'
   12004 'large cell' 'Big and beautiful'
```

**Adding a new loop packet.** PyCifRW does not (yet) directly support this: the following code shows one way to accomplish this indirectly for the above example.

```
newdata= {'_example':['101.1','255'],
 '_example_detail':['medium cell','also medium'],
 '_comment':['manageable','still manageable']
         }
olddata = cb.GetLoop('_example') #(key,value) list
map(lambda a:newdata[a[0]].extend(a[1]),loopdata)
cb.AddCifItem((newdata.keys(),newdata.values()))
```

Note that, as the lists returned by PyCIFRW are direct pointers to the original lists, it is possible to extend them directly (e.g. `cb['_example'].append('101.1')`), however, this bypasses all data value syntax checks and loop length checks and is not recommended.

## 2.3 Writing Cif Files

The `CifFile` method `WriteOut` returns a string which may be passed to an open file descriptor:

```
>>>outfile = open("mycif.cif")
>>>outfile.write(cf.WriteOut())
```

An alternative method uses the built-in Python `str()` function:

```
>>>outfile.write(str(cf))
```

`WriteOut` takes an optional argument, `comment`, which should be a string containing a comment which will be placed at the top of the output file. This comment string must already contain # characters at the beginning of lines:

```
>>>outfile.write(cf.WriteOut("#This is a test file"))
```

There is currently no way to easily specify the order of output of items within a `CifFile` or `CifBlock`.

# 3 Dictionaries and Validation

## 3.1 Dictionaries

DDL dictionaries may also be read into `CifFile` objects. For this purpose, `CifBlock` objects automatically support save frames (used in DDL2 dictionaries), which are accessed using the "`saves`" key. The value of this key is a collection of `CifBlock` objects indexed by save frame name, and available operations are similar to those available for a `CifFile`, which is also a collection of `CifBlocks`.

A `CifDic` object hides the difference between DDL1 dictionaries, where all definitions are separate data blocks, and DDL2 dictionaries, where all definitions are in save frames of a single data block. A `CifDic` is initialised with a single file name or `CifFile` object:

```
cd = CifFile.CifDic("cif_core.dic")
```

Definitions are accessed using the usual notation, e.g. `cd['_atom_site_aniso_label']`. Return values are always `CifBlock` objects. Additionally, the `CifDic` object contains a number of instance variables derived from dictionary global data:

**dicname** The dictionary name + version as given in the dictionary

**diclang** 'DDL1' or 'DDL2'

**typedic** Python dictionary matching typecode with compiled regular expression

`CifDic` objects provide a large number of validation functions, which all return a Python dictionary which contains at least the key "`result`". "`result`" takes the values `True`, `False` or `None` depending on the success, failure or non-applicability of each test. In case of failure, additional keys are returned depending on the nature of the error.

## 3.2 Validation

A top level function is provided for convenient validation of CIF files:

```
CifFile.validate("mycif.cif",dic = "cif_core.dic")
```

This returns a tuple `(valid_result, no_matches)`. `valid_result` and `no_matches` are Python dictionaries indexed by block name. For `valid_result`, the value for each block is itself a dictionary indexed by `item_name`. The value attached to each item name is a list of `(check_function, check_result)` tuples, with `check_result` a small dictionary containing at least the key "`result`". All tests which passed or were not applicable are removed from this dictionary, so result is always `False`. Additional keys contain auxiliary information depending on the test. Each of the items in `no_matches` is a simple list of item names which were not found in the dictionary.

If a simple validation report is required, the function `validate_report` can be called on the output of the above function, printing a simple ASCII report. This function can be studied as an example of how to process the complex structure returned by the '`validate`' function.

### 3.2.1 Limitations on validation

1. (DDL2 only) When validating data dictionaries themselves, no checks are made on group and subgroup consistency (e.g. that a specified subgroup is actually defined).

2. (DDL1 only) Some _type_construct attributes in the DDL1 spec file are not machine-readable, so values cannot be checked for consistency

## 3.3 ValidCifFile objects

A `ValidCifFile` object behaves identically to a `CifFile` object with the additional characteristic that it is valid against the given dictionary object. Any attempt to set a data value, or add or remove a data name, that would invalidate the object raises a `ValidCifFile` error.

Additional keywords for initialisation are:

**dic** A `CifDic` object to use in validation

**diclist** A list of `CifFile` objects or filenames to be merged into a `CifDic` object (see below)

**mergemode** Choose merging method (one of '`strict`','`overlay`', '`replace`')

## 3.4 Merging dictionaries

PyCIFRW provides a top-level function to merge DDL1/2 dictionary files. It takes a list of CIF filenames or `CifFile` objects, and a `mergemode` keyword argument. CIF files are merged from left to right, that is, the second file in the list is merged into the first file in the list and so on.

For completeness we list the arguments of the `CifFile merge` method:

**new_block_set** (first argument, no keyword) The new dictionary to be merged into the current dictionary

**mode** merging mode to use ('strict', 'overlay' or 'replace')

**single_block** a two element list `[oldblockname, newblockname]`, where `oldblockname` in the current file is merged with `newblockname` in the new file. This is useful when blocknames don't match

**idblock** This block is ignored when merging - useful when merging DDL1 dictionaries in strict mode, in which case the `on_this_dictionary` block would cause an error.

### 3.4.1 Limitations on merging

In overlay mode, the COMCIFS recommendations require that, when both definitions contain identical attributes which can be looped, the merging process should construct those loops and include both sets of data in the new loop.

This is not yet implemented in PyCIFRW, as it involves checking the DDL1/DDL2 spec to determine which attributes may be looped together.

# 4 Working with STAR files

## 4.1 Creating STAR files

Star files are created entirely analogously to CIF files, using the `StarFile` object or `ReadStar` function.

## 4.2 Manipulating values

The usual square bracket notation applies, as for `CifFile` and `CifBlock` objects. `StarFiles` are built out of `StarBlock` objects in exactly the same way as `CifFile` objects are built out of `CifBlock` objects. `StarBlock` objects can contain any number of `LoopBlock` objects, which represent STAR loop blocks. Crucially, these `LoopBlock` objects may contain nested loops, which are also `LoopBlock` objects. Loops are inserted into a `LoopBlock` by calling the `insert_loop` method, and may be nested to an arbitrary level.

### 4.2.1 Iterators

Any LoopBlock object has two iterator methods: `recursive_iter` and `flat_iterator`. On each call of the iterator created by a `recursive_iter` call, a Python dictionary is returned with single-valued keys corresponding to a single set of values. If there are multiple trees of nested loops in a `LoopBlock`, each tree is iterated over separately, as there is no reason that looped values inside a second loop block would have any relationship with values inside a first loop block. This iterator will thus return all possible sets of values for the `LoopBlock`.

The `flat_iterator` method does not dig down into nested loops. Instead, iterators created from it return a new `LoopBlock` with key-value pairs corresponding to a single top-level packet; nested loops are included, but they also have only data corresponding to the selected top-level packet available. This iterator thus iterates through the top-level packets, collapsing the nesting level by one.

# 5 Example programs

A program which uses PyCIFRW for validation, `validate_cif.py`, is included in the distribution in the Programs subdirectory. It will validate a CIF file (including dictionaries) against one or more dictionaries which may be specified

by name and version or as a filename on the local disk. If name and version are specified, the IUCr canonical registry or a local registry is used to find the dictionary and download it if necessary.

### 5.1 Usage

```
python validate_cif.py [options] ciffile
```

### 5.2 Options

–**version** show version number and exit

**-h,–help** print short help message

**-d dirname** directory to find/store dictionary files

**-f dictname** filename of locally-stored dictionary

**-u version** dictionary version to resolve using registry

**-n name** dictionary name to resolve using registry

**-s** store downloaded dictionary locally (default True)

**-c** fetch and use canonical registry from IUCr

**-r registry** location of registry as filename or URL

**-t** The file to be checked is itself a DDL2 dictionary

## 6 Further information

The source files are in a literate programming format (noweb) with file extension .nw. HTML documentation generated from these files and containing both code and copious comments is included in the downloaded package. Details of interpretation of the current standards as relates to validation can be found in these files.