```
 data  save
block  block  categ. item
----------------------------

cif_img.dic
        -
                _category_group_list
                        _category_group_list.description
                        _category_group_list.id
                        _category_group_list.parent_id
                _dictionary
                        _dictionary.datablock_id
                        _dictionary.title
                        _dictionary.version
                _dictionary_history
                        _dictionary_history.revision
                        _dictionary_history.update
                        _dictionary_history.version
                _item_type_list
                        _item_type_list.code
                        _item_type_list.construct
                        _item_type_list.detail
                        _item_type_list.primitive_code
                _item_units_conversion
                        _item_units_conversion.factor
                        _item_units_conversion.from_code
                        _item_units_conversion.operator
                        _item_units_conversion.to_code
                _item_units_list
                        _item_units_list.code
                        _item_units_list.detail
        ARRAY_DATA
                _category
                        _category.description
                        _category.id
                        _category.mandatory_code
                _category_examples
                        _category_examples.case
                        _category_examples.detail
                _category_group
                        _category_group.id
                _category_key
                        _category_key.name
```

Fig. 5.3.6.2. Structure of the imgCIF dictionary (Chapter 4.6) as described by the *get_keys* method of the *STAR::DataBlock* module. Only the high-order file structure and the contents of the first category are included in this extract.

5.3.6.1.5. *STAR::Writer and STAR::Filter*

Two other modules are supplied by this package. *STAR::Writer* is a prototype module that can write STAR::DataBlock objects out as files in different formats; currently only the *write_cif* method exists to output a conformant CIF. *STAR::Filter* is an interactive module that prompts the user to select or reject individual categories from a STAR::Dictionary object when building a subset of the larger dictionary.

5.3.6.2. *PyCifRW*: CIF reading and writing in Python

PyCifRW (Hester, 2006) is a simple CIF input/output utility written in Python. It does not validate content against dictionaries, but it does provide a robust parser that has been extensively tested against various test files containing subtle syntactic features and against the collection of over 18 000 macromolecular CIFs available from the Protein Data Bank. The parser was implemented using the *Yapps2* parser generator (Patel, 2002) and is based on the draft Backus–Naur form (BNF) developed during a community exercise to review the CIF specification of Chapter 2.2.

As with the Perl library discussed above, *PyCifRW* presents an object-oriented set of classes and methods. Two classes are provided, *CifFile* and *CifBlock*.

A *CifFile* object provides an associative array of *CifBlock* objects, accessed by data-block name.

The methods available for the *CifFile* type are: *ReadCif(filename)*, which initializes or reinitializes a file to contain the CIF contents; *GetBlocks()*, which returns a list of the data-block names in the file; *NewBlock(blockname, [block contents])*, which adds a new data block to the file object; and *WriteOut(comment)*, which returns the contents of the current file as a CIF-conformant string, with an optional comment at the beginning. For the *NewBlock* method, the optional *block contents* must be a valid *CifBlock* object, as described below. The *NewBlock* method returns the name of the new block created, which will *not* be the requested *blockname* if a data block of the same name already exists (this conforms to the STAR and CIF requirement that data-block names must be unique within a file).

A *CifBlock* object represents the contents of a data block. The methods available to retrieve or manipulate the contents are: *GetCifItem(itemname)*, which will return the value of the data item with data name given by *itemname* (and which can be a single value or an array of looped values); *AddCifItem(data)*, which adds *data* to the current block, where *data* represents either a data name and an associated single value, or, for the case of looped data, a tuple containing an array of data names and an array of arrays of associated data values; *RemoveCifItem(dataname)*, to remove the specified data item from the current block; and *GetLoop(dataname)*, which returns a list of all data items occurring in the loop containing the data name provided. If *dataname* does not represent a looped data item, an error is returned.

The *GetLoop* method is important for the proper handling of looped data, and care is taken to handle loops robustly and efficiently. Items that are initially looped together are kept in the same loop structure.

Both *CifFile* and *CifBlock* objects act as Python mapping objects, which has the advantage that the value of a data item can be read or changed using an intuitive square-bracket notation. For example, a program can retrieve the value of a data item named `_my_data_item_name` in block 'myblockname' of a previously opened file *cf* using the following syntax:

```
value = cf["myblockname"]["_my_data_item_name"]
```

The returned value is either a single item, or a Python array of values if the data item occurs in a loop. Values are set in an analogous way and other common operations with mapping objects are also implemented.

5.3.7. Rapid development tools

The programs described so far in this chapter tend to fulfil a single purpose. Each program addresses a single clearly defined task and is of benefit to a user who has no need or desire to write a customized program. Where there is a need to write a new application, libraries of subroutines are available to the full-time programmer, such as those described in Chapters 5.4 to 5.6. However, in between these extremes, there are a large number of cases for which there is a need to combine the functionality of a number of existing programs without incurring the overhead of writing a new integrated application.

This section describes utilities that assist the development of new applications from existing software.

references