Journal of
Applied Crystallography
ISSN 0021-8898

Received 28 March 2006
Accepted 28 April 2006

A validating CIF parser: *PyCIFRW*

J. R. Hester

ASRP, ANSTO, PMB 1, Menai, NSW, 2234, Australia. Correspondence e-mail: jrh@anbf2.kek.jp

PyCIFRW is a general-purpose Python package providing a simple, powerful interface for working with CIF files. Objects and methods are available for reading, writing and manipulating CIF files and dictionaries. Comprehensive validation of CIF files and dictionaries against DDL1 or DDL2 dictionaries is also possible. *PyCIFRW* is easily included in larger projects and is portable across a large number of platforms. Although written in an interpreted language, parsing and validation times are acceptable for most applications.

1. Introduction

The CIF syntax (Hall *et al.*, 1991) for data archiving and exchange is well established in crystallography. An important feature of the CIF format is the availability of standard dictionaries which codify the meanings of discipline-specific sets of CIF data items in both human and machine-readable form. These dictionaries are written using a dictionary definition language (DDL) (Hall & Cook, 2005; Westbrook *et al.*, 2005) which is itself simply a set of standard CIF tags. The machine-readable tag–value pairs in these dictionaries describe conditions that each CIF data item should satisfy; for example, the DDL attributes for a particular data item may restrict values to positive integers, or allow this data item to occur only in a loop with another specific data item.

Programming libraries for working with CIF and STAR files (CIF syntax is a restricted form of STAR syntax) significantly ease the task of adding CIF functionality to software projects. Such libraries have long been available in the well established compiled languages Fortran (Hall & Bernstein, 1996) and C, or variants (Westbrook *et al.*, 1997; Chang & Bourne, 1998). Among other capabilities, these libraries read and write syntactically correct CIF files, and in some cases check CIF data values and structures against one or more DDL dictionaries.

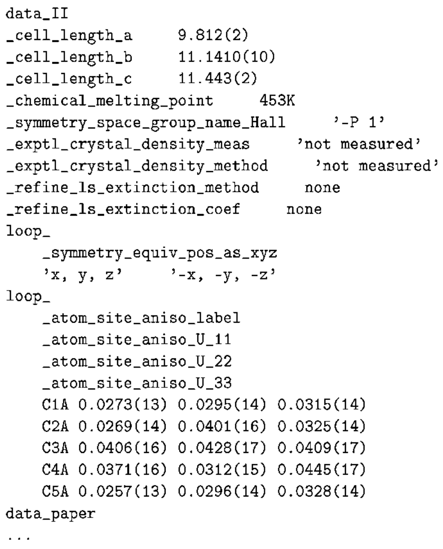
Such broad CIF support is generally lacking for interpreted languages, despite a number of advantages that these languages have over compiled languages. These advantages include: an identical programming interface on a variety of the common desktop and mainframe platforms, allowing creation of portable programs with no additional effort from the programmer; dynamic typing and high-level constructs mean that similar programming tasks require significantly fewer lines of code than their C/Fortran equivalent, leading to better program maintainability; simple tasks can be accomplished using built-in interactive interpreters; and program distribution and installation is usually simpler for both author and installer. Such advantages come at the cost of a much slower execution speed and larger memory footprint during execution. However, with the rise of desktop computer power over the past decade, these costs have diminished to the point where interpreted languages have become viable alternatives for performing common computational tasks. Among the most established of these languages are Tcl (Ousterhout, 1994), Perl (Wall *et al.*, 2000) and Python (van Rossum & Drake, 2003). A degree of CIF parsing and validation support in these languages is provided by *HICCuP* (Edgington, 1997), written in Python, and *STAR::Parser* (Bluhm, 2000), written in Perl. *HICCuP* was an early stand-alone application for working with CIFs, and offered a series of validation tests against preprocessed DDL1 dictionaries. *STAR::Parser* is a set of modules offering both parsing and validity checking against DDL1 and DDL2 dictionaries.

The project described here is a comprehensive native Python library for CIF parsing and validating. It differs from the above efforts in the broader scope of its validation tests against both DDL1 and DDL2 dictionaries and the ability to work with multiple merged dictionaries. Unlike *HICCuP*, it is a general purpose, native implementation designed for rapid integration into larger projects.

2. Using *PyCIFRW*

This section gives a sample of the simple and powerful interface provided by *PyCIFRW*, using the example CIF file of Fig. 1.

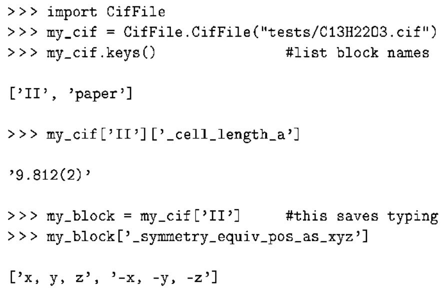
An important task for a CIF application is to extract data from a pre-existing CIF file. Fig. 2 shows an interactive session reading in a file and accessing data blocks. A CIF file is read in by passing a file name or URL when initializing a `CifFile` object. Data blocks within the file and data items within the data blocks are then accessed using square-bracket notation, emulating the syntax for access into the



```
data_II
_cell_length_a     9.812(2)
_cell_length_b     11.1410(10)
_cell_length_c     11.443(2)
_chemical_melting_point     453K
_symmetry_space_group_name_Hall     '-P 1'
_exptl_crystal_density_meas     'not measured'
_exptl_crystal_density_method     'not measured'
_refine_ls_extinction_method     none
_refine_ls_extinction_coef     none
loop_
    _symmetry_equiv_pos_as_xyz
    'x, y, z'     '-x, -y, -z'
loop_
    _atom_site_aniso_label
    _atom_site_aniso_U_11
    _atom_site_aniso_U_22
    _atom_site_aniso_U_33
    C1A 0.0273(13) 0.0295(14) 0.0315(14)
    C2A 0.0269(14) 0.0401(16) 0.0325(14)
    C3A 0.0406(16) 0.0428(17) 0.0409(17)
    C4A 0.0371(16) 0.0312(15) 0.0445(17)
    C5A 0.0257(13) 0.0296(14) 0.0328(14)
data_paper
...
```

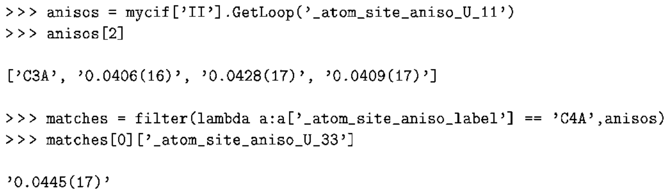
Figure 1
CIF file used in later examples, adapted from a CIF deposited for $C_{13}H_{22}O_3$ (Mondal *et al.*, 2002).

doi:10.1107/S0021889806015627



```
>>> import CifFile
>>> my_cif = CifFile.CifFile("tests/C13H22O3.cif")
>>> my_cif.keys()              #list block names

['II', 'paper']

>>> my_cif['II']['_cell_length_a']

'9.812(2)'

>>> my_block = my_cif['II']     #this saves typing
>>> my_block['_symmetry_equiv_pos_as_xyz']

['x, y, z', '-x, -y, -z']
```

Figure 2
Example of interactive use, showing retrieval of non-looped data item `_cell_length_a`, which is returned as a string, and retrieval of `_symmetry_equiv_pos_as_xyz` from the `loop_` block, which is returned as a Python list object.



```
>>> anisos = mycif['II'].GetLoop('_atom_site_aniso_U_11')
>>> anisos[2]

['C3A', '0.0406(16)', '0.0428(17)', '0.0409(17)']

>>> matches = filter(lambda a:a['_atom_site_aniso_label'] == 'C4A',anisos)
>>> matches[0]['_atom_site_aniso_U_33']

'0.0445(17)'
```

Figure 3
Working with looped data. A loop block is selected and the third packet (out of five; see Fig. 1) accessed. Note that Python list indices start from 0. The U_{33} value for atom C4A is then found using an alternative access method.

built-in Python 'dictionary' data type. This emulation includes a series of standard Python dictionary access methods for extracting and setting values.

Each CIF data block (*e.g.* `mycif['II']` in Fig. 2) is an instance of a `CifBlock` object. Data values for items contained in a `CifBlock` are either strings or numbers. If the value of a looped item name is requested, a Python list of strings or numbers is returned; however, a complete set of looped items is typically more useful than lists of a single item's values. For this reason, the `CifBlock` method `GetLoop(loop_dataname)` returns a `CifLoopBlock` object representing the CIF loop block containing `loop_dataname`. As a `CifLoopBlock` object represents an entire loop it is possible to add, remove and alter co-looped items by simple assignment using the square-bracket notation.

A CIF loop is composed of one or more loop 'packets', where a single loop packet contains the values taken by all of the data items in one iteration of the loop. `CifLoopBlock` objects allow retrieval of a packet from a loop block by packet number, as shown in Fig. 3. In this style of access, the link between position in the returned Python list and data name is obtained by calling method `GetItemOrder()`. More usefully, Python-style iterators are also defined, allowing economical selection of packets satisfying an arbitrary condition. The final command in Fig. 3 demonstrates the use of this feature. The `filter` command will return only those packets in the second argument for which the function given in the first argument returns `True`. When executing the function for each packet in `CifLoopBlock` object `anisos`, the packet will be passed as the first variable to the function; in the example, an anonymous ('lambda') function of one variable (`a`) is specified. This function checks whether the variable's `_atom_site_aniso_label` is equal to '`C4A`', thereby selecting only packets for atom C4A.

These loop methods and iterators are also available for the more general case of STAR nested loops (see §3 below).

2.1. Save frames

While CIF data files should not use save frames, *PyCIFRW* reads, manipulates and writes save frames in order to support DDL2 dictionaries. The save frames in a CIF data block are available through the special key '`saves`', the value of which is also a dictionary. The keys in this dictionary are the frame names. Each save frame behaves identically to a normal data block.

For example, if `df = CifFile('cif_mm_2.0.03.dic')`, then `df['cif_mm.dic']['saves'].keys()` will give a list of saveframenames (in this case, data and category name definitions from the macromolecular CIF dictionary), and `df['cif_mm.dic']['saves']['atom_site']` will be a `CifBlock` containing the attributes of the `atom_site` category. Save frames are invisible during normal operations on a data block: for example, `df['cif_mm.dic'].keys()` returns a short list of dictionary global data names, excluding the 1800 data name definitions found inside the save frames.

2.2. CIF output

A string object suitable for writing to a file is obtained by calling the built-in Python `str` function on a `CifFile` object. Internal formatting functions insert quotation marks and semicolons where necessary, and illegal characters will be absent, as the presence of such characters in a data value would have caused an error to be raised when the data value was first set. Overlong lines are broken at the last whitespace before the line length limit, or, if no whitespace is available, at an arbitrary 80 characters. Special handling of long lines using the backslash convention is not yet implemented.

2.3. Validation

In the following, 'validation' is used to mean checking that the values and placement of data items in a CIF data block conform to the specifications contained in one or more machine-readable DDL dictionaries.

PyCIFRW defines a `validate` function which returns validation results for the given CIF file and CIF dictionary or dictionaries. Alternatively, *PyCIFRW* bundles a simple command-line program called `cif_validate.py` which executes all relevant validation functions on the given CIF file when passed a list of dictionaries. Dictionaries may be specified by name and version, or by file name, and are downloaded if necessary. An example use of this program is shown in Fig. 4. DDL2 dictionaries may be used to validate DDL1-style CIF files; this is achieved by creating new dictionary entries during internal initialization using the value of the DDL2 `_item_aliases.alias_name` attribute.

A list of the validation tests performed by *PyCIFRW* is given in Table 1. Each test was developed by examining the description of each attribute in the DDL1 and DDL2 specifications published in the *International Tables for Crystallography*, Vol. G (Hall & McMahon, 2005), where necessary referring to canonical dictionaries for examples of correct attribute use. Note that, while an output file produced by *PyCIFRW* is guaranteed to be syntactically correct, it is not guaranteed to be valid unless the `validate` function returns no errors.

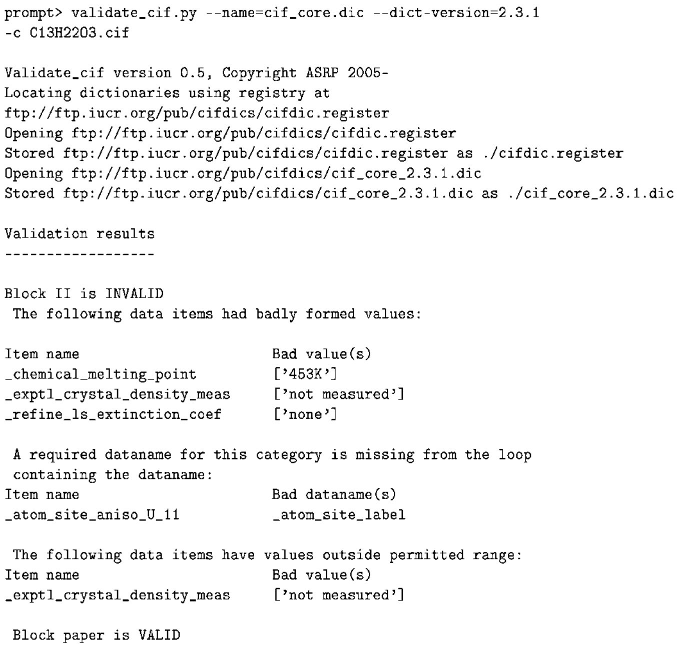
2.3.1. Treatment of multiple dictionaries. When multiple dictionaries are provided to the validation routines, they are first merged according to the protocol suggested by McMahon (2005) (BM), with the following variations.

(*a*) The calling function is responsible for ordering the dictionary list [BM step (i)].

Table 1
Validation tests in *PyCIFRW*.

NA = not applicable.

	Relevant attributes	
Test	DDL1	DDL2
Check against type specified in dictionary	`_type`,`_type_construct`	`_item_type.code` and `_item_type_list.construct`
Check value is in allowed range	`_enumeration_range`	`_item_range.maximum` and `_item_range.minimum`
Check value is in allowed set	`_enumeration`	`_item_enumeration.value`
Check that e.s.d.'s are allowed	`_type_conditions`	`_item_type_conditions.code`
Check that value is/is not looped	`_list`	NA
Check that all items are from same category	`_category`	`_item.category_id`
Check that mandatory items are present in the loop	`_list_mandatory`	`_item.mandatory_code`
Check that any reference names specified by loop items are present	`_list_reference`	NA
Check that child items only take values of the parent item and that parent is present	`_list_link_child`	`_item_linked.child_name`
Check that all items which this item depend on are present in the data block	NA	`_item_dependent.dependent_name`
Check that at least one item from any mandatory categories is present	NA	`_category.mandatory_code`
Check that sets of items take unique values	`_list_uniqueness`	`_category_key.name`



```
prompt> validate_cif.py --name=cif_core.dic --dict-version=2.3.1
-c C13H22O3.cif

Validate_cif version 0.5, Copyright ASRP 2005-
Locating dictionaries using registry at
ftp://ftp.iucr.org/pub/cifdics/cifdic.register
Opening ftp://ftp.iucr.org/pub/cifdics/cifdic.register
Stored ftp://ftp.iucr.org/pub/cifdics/cifdic.register as ./cifdic.register
Opening ftp://ftp.iucr.org/pub/cifdics/cif_core_2.3.1.dic
Stored ftp://ftp.iucr.org/pub/cifdics/cif_core_2.3.1.dic as ./cif_core_2.3.1.dic

Validation results
------------------

Block II is INVALID
 The following data items had badly formed values:

Item name                       Bad value(s)
_chemical_melting_point         ['453K']
_exptl_crystal_density_meas     ['not measured']
_refine_ls_extinction_coef      ['none']

 A required dataname for this category is missing from the loop
 containing the dataname:
Item name                       Bad dataname(s)
_atom_site_aniso_U_11           _atom_site_label

 The following data items have values outside permitted range:
Item name                       Bad value(s)
_exptl_crystal_density_meas     ['not measured']

 Block paper is VALID
```

Figure 4
Example of validation output using small-molecule CIF `C13H22O3.cif` (Mondal *et al.*, 2002).

(*b*) Contrary to BM, when a dictionary definition contains unlooped attributes, and the new dictionary to be merged with it also contains some or all of these attributes, 'overlay' mode will not attempt to construct a loop including both old and new attributes, even if those attributes may be looped. Instead, the merged dictionary will contain the attribute as it appears in the new dictionary, if necessary removing it from any loop block or adding the complete loop block in which it appears. Behaviour as specified by BM requires access to the DDL1/2 dictionaries specifying looping properties for the data items, and will be implemented in a future release.

2.3.2. Comments on the validation of DDL1 and DDL2 dictionaries. Just as CIF data files are subject to constraints expressed in CIF dictionaries, CIF dictionaries are themselves subject to constraints expressed in the dictionaries that define the DDL1 and DDL2 attributes. The use of save frames in DDL2 dictionaries sometimes leads to subtle differences in interpretation of attribute meaning when validating dictionaries compared with validating CIF data files. In particular, it becomes important to identify the correct object for validation: in the general case, we conclude that it is the complete data block which is valid or invalid, rather than, for example, a single definition block. While it might be expected that the combination of enclosing data block and single definition save frame for a DDL2 dictionary would be sufficient to satisfy validity constraints, this is also not generally true. This behaviour arises because ascertaining the correctness of a number of DDL2 attributes requires checking the presence or value of certain attributes which can only be found in other save frames. For example, `_item.category_id`, which appears in most save frames, is a child of `_category.id`, so values taken by `_category.id` in category-definition save frames must be examined to check that `_item.category_id` takes legal values.

The interpretation of the `_category.mandatory_code` attribute is somewhat unclear in the context of a DDL2 dictionary. Vol. G of *International Tables for Crystallography* (Hall & McMahon, 2005) states that this attribute 'specifies whether the category must appear in any data block based on this dictionary' (p. 64). Uncertainty arises from the setting of this attribute to `yes` for the `ITEM_DESCRIPTION` category. In currently available DDL2 dictionaries, attributes belonging to this category appear in those save frames that define data item names. The intention of the authors of the `ITEM_DESCRIPTION` category definition in specifying that this is a mandatory category would appear to have been to force all defined names to have an associated description, which implies from a validation point of view that the scope of a search when checking for the presence of such a mandatory category item is a single save frame. However, under such an interpretation, all category definition save frames in DDL2 dictionaries would be non-conformant, as they do not contain any data items from the `ITEM_DESCRIPTION` category. Therefore, an alternative interpretation is adopted, where an item is considered to appear in a data block even if the only appearance is in one or more save frames.

As implied by the above discussion, *PyCIFRW* will correctly validate DDL1 dictionaries in the same way as ordinary CIF files; however, DDL2 dictionaries require a special flag to the validation routines. This flag causes *PyCIFRW* firstly to make the translation save frame → data block for the purposes of validation, but then to search outside a single save frame to resolve parent–child references and check for the presence of items from mandatory categories in the dictionary as a whole.

3. Implementation

The various *PyCIFRW* objects are built out of two fundamental objects defined in the underlying STAR file implementation: `BlockCollection` and `LoopBlock`.

`LoopBlock` objects are collections of key–value pairs with a special key '`loops`' containing a possibly empty list of `LoopBlock`s, corresponding to nested loops. This list is also searched when retrieving or setting data values, so that data names act as if they are keys of the outermost `LoopBlock`. `LoopBlock`s are STAR-conformant objects; in particular, they may be arbitrarily deeply nested, and have no name length restrictions. A number of methods are provided in the underlying STAR file implementation for iterating over nested loop packets, some of which are useful in the CIF context, as described in the previous section.

A `CifLoopBlock` is a `LoopBlock` with restricted-length key names, and values which are either simple lists or atomic values. A `CifBlock` is a `CifLoopBlock` with non-list data values. The underlying generalization is that a STAR data block is a special case of a STAR loop block; the set of all non-looped key–value pairs in a CIF or STAR block could be equally well expressed as a loop with a single data packet.

A `BlockCollection` object represents a collection of objects derived from `LoopBlock` and is used to construct the `CifFile` object and each block's set of save frames, which are also collections of blocks. `BlockCollections` add case-insensitivity of key names and preservation of input order to the standard Python dictionary type.

3.1. CIF input

The data representation described earlier is built up during parsing of a CIF file. Parsing is accomplished using a parser constructed by the Yapps2 lexer/parser (Patel, 2003) from a simple implementation of the STAR grammar specification. This grammar, together with pre- and post-parsing checks, is designed to be rigorously conformant to the CIF 1.1 standard (Hall *et al.*, 2005) and as such *PyCIFRW* can be used as a CIF syntax checker. It has been tested against the IUCr 'trip' test suite (http://www.iucr.org/iucr-top/cif/developers/trip) and correctly identifies both conformant and non-conformant files.

3.2. Dictionaries, validation and merging

The `CifDic` object is subclassed from a `BlockCollection` object, and requires one or more DDL1 and/or DDL2 dictionaries to be provided at initialization. These dictionaries are normalized to a uniform internal structure, allowing data name definitions to be accessed using square bracket notation instead of needing to access the various save frames in the case of DDL2.

The validation routines listed in Table 1 are methods of the `CifDic` object. CIF value type checking for DDL2 dictionaries is performed by direct use of the regular expressions contained in the dictionary file.

A number of obvious transformations are performed when initializing DDL1 dictionaries in order to produce DDL2-like behaviour: category-wide information (*e.g.* `_list_mandatory`, `_list_uniqueness`) is transferred into a category block; definitions containing looped `_name` data items are expanded to include one item name per definition; enumeration ranges are expressed using DDL2-style maximum/minimum specifications; and specific `_type_construct` attributes are moved to the dictionary global level. DDL2 dictionaries are also transformed by moving parent/child attributes to the corresponding data name definition as for DDL1 dictionaries. Due to the difficulty of reversing this latter transformation, DDL dictionary merging in *PyCIFRW* is performed at the `CifFile` level, and the final merged `CifFile` object is used to initialize the `CifDic` object.

Table 2
Execution times on a 1.6 GHz IBM PC with 512 kbyte RAM running Linux 2.6.

Validation time does not include dictionary parsing time. File `1YGG.cif` was obtained from the Protein Data Bank, ID 1YGG (Leduc *et al.*, 2005).

File name	File size (kbyte)	Parsing time (s)	Validation time (s)
`1YGG.cif`	484	28	7
`C13H22O3.cif`	28	1	0.4
`cif_core.dic`	444	7	–
`cif_mm_2.0.03.dic`	1581	48	–

4. Discussion

Table 2 gives some representative times for input and validation of typical small-molecule and protein data files, as well as dictionary preparation times (which are dominated by the parsing stage). As expected for an interpreted language, parsing of typical input files is of the order of seconds with current hardware, compared with compiled-language parsers which would take small fractions of a second. Validation times give an order of magnitude estimate of data access times for data-hungry applications, as validation requires accessing every data item at least once.

For applications which do not access large numbers of CIF files, these times are well within acceptable limits. As compensation for the loss in execution efficiency, programmers obtain maintainable, easily distributable code which runs without change on all platforms, and end users have the option of simple command-line interaction with CIF files.

5. Availability

PyCIFRW runs on all platforms supported by Python, which includes Windows, Linux and Mac OS X. The program code and developer documentation are produced from single files in the literate programming *noweb* (Ramsey, 1994) format to ensure maximum accessibility and maintainability. *PyCIFRW* is copyright the Australian Synchrotron Research Program and is freely downloadable under liberal licensing terms from http://anbf2.kek.jp/CIF. It is also bundled as part of the *CCTBX* project (Grosse-Kunstleve *et al.*, 2002).

The author is grateful to a number of early users of *PyCIFRW*, especially R. Grosse-Kunstleve and D. du Boulay.

References

Bluhm, W. (2000). Star (CIF) parser, http://pdb.sdsc.edu/STAR/index.html.

Chang, W. & Bourne, P. E. (1998). *J. Appl. Cryst.* **31**, 505–509.

Edgington, P. R. (1997). *HICCuP*: *High-Integrity CIF Checking Using Python*, Cambridge Crystallographic Data Centre, UK.

Grosse-Kunstleve, R. W., Sauter, N. K., Moriarty, N. W. & Adams, P. D. (2002). *J. Appl. Cryst.* **35**, 126–136.

Hall, S. R. & McMahon, B. (2005). Editors. *International Tables for Crystallography*, Vol. G. IUCr/Springer.

Hall, S. R., Allen, F. H. & Brown, I. D. (1991). *Acta Cryst.* A**47**, 655–685.

Hall, S. R. & Bernstein, H. J. (1996). *J. Appl. Cryst.* **29**, 598–603.

Hall, S. R. & Cook, A. P. F. (2005). *International Tables for Crystallography*, Vol. G, edited by S. R Hall & B. McMahon, ch. 2.5, pp. 53–60. IUCr/Springer.

Hall, S. R., Spadaccini, N., Brown, I. D., Bernstein, H. J., Westbrook, J. D. & McMahon, B. (2005). *International Tables for Crystallography*, Vol. G, edited by S. R Hall & B. McMahon, ch. 2.2.7, pp. 25–36. IUCr/Springer.

Leduc, Y. A., Prasad, L., Laivenieks, M. J. G. Z. & Delbaere, L. T. (2005). *Acta Cryst.* D**61**, 903–912.

McMahon, B. (2005). *International Tables for Crystallography*, Vol. G, edited by S. R Hall & B. McMahon, ch. 3.1.9, pp. 88–89. IUCr/Springer.

Mondal, S., Mukherjee, M., Roy, A., Mukherjee, D. & Helliwell, M. (2002). *Acta Cryst.* C**58**, o474–o476.

Ousterhout, J. K. (1994). *The Tcl and the Tk Toolkit*, London: Addison-Wesley.

Patel, A. (2003). Parsing with yapps. http://theory.stanford.edu/~amitp/Yapps.

Ramsey, N. (1994). *IEEE Softw.* **11**(5), 97–105.

Rossum, G. von & Drake, F. L. Jr (2003). *The Python Language Reference Manual.* Bristol: Network Theory Ltd.

Wall, L., Christiansen, T. & Orwant, J. (2000). *Programming Perl.* 3rd ed. Cambridge, MA: O'Reilly.

Westbrook, J. D., Berman, H. M. & Hall, S. R. (2005). *International Tables for Crystallography*, Vol. G, edited by S. R Hall & B. McMahon, ch. 2.6, pp. 61–70. IUCr/Springer.

Westbrook, J. D., Hsieh, S.-H. & Fitzgerald, P. M. D. (1997). *J. Appl. Cryst.* **30**, 79–83.