The GAVO VOTable Library

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2023-02-09

A library to process VOTables using python

Author: Markus Demleitner

Email: gavo@ari.uni-heidelberg.de

Contents

Introduction

This library lets you parse and generate VOTables. These are data containers as used in the Virtual Observatory and specified in http://www.ivoa.net/Documents/VOTable/20091130/REC-VOTable-1.2.html

This library supports Version 1.3 VOTables (including BINARY3) and to some extent earlier versions as well.

There are many other libraries to parse VOTables into python programs, the default probably being astropy's. This one is geared for maximal control, streaming, and the possibility to write fairly mad instance documents.

There's a simplified high-level interface for simple applications, too.

Obtaining the library

Current versions of the library are available from DaCHS distribution page. Users of Debian stable (and similar distributions not too far removed), will want to pull the library from the data center's apt archive; see the DaCHS install docs.
This library is part of the GAVO Data Center Helper Suite.

Simple parsing

To parse simple (one-table), moderately sized VOTable, do:

```
votable.load(source) -> data, metadata
```

Here, `data` is a list of records, each of which is a sequence, `metadata` is a `TableMetadata` instance, and `source` can be a string naming a local file, or it can be a file-like object. To parse material in a string, use `votable.loads`.
`votable.load` only looks at the first TABLE element encountered. If the VOTable does not contain any tabular data at all (e.g., error messages from various VO protocols), `(None, None)` is returned.
Metadata contains is the VOTable.TABLE element the data comes from in its votTable atttribute. By iterating over the metadata you get the field objects. For example:

```
from gavo import votable
labels = [field.name for field in metadata]
print [dict(zip(labels, row)) for row in data]
```

There's a convenience method that does the dictification for you; to iterate over all rows of a VOTable as dicts, you can say:

```
data, metadata = votable.load(source)
for row in metadata.iterDicts(data):
   ...
```

If you want to create a numpy record array from that data, you can say:

```
data, metadata = votable.load(source)
ra = rec.array(data, dtype=votable.makeDtype(metadata))
```

However, you cannot in general store NULL values in record arrays (as None, that is), so this code will fail for many tables unless one introduces proper null values (e.g., nan for floats; for ints, you could ask metadata for the null value used by the VOTable). Also, record arrays cannot store variable-length strings, so makeDtype assumes some default length. Pass a *defaultStringLength* keyword argument if your strings get truncated (or replace the `*` in the `FIELD` object with whatever actual length you deem sufficient).
All the above examples will fail on VOTables with features they cannot understand. Current VO practices recommend ignoring unknown tags and attributes, though. The VOTable library has a brute force approach so far; pass `raiseOnInvalid=False` to the parsing functions and they will ignore constructs the do not understand. Note that this will lead to surprising behaviour in cases where you input non-VOTables to your program. As long as it is well-formed XML, you will not receive error messages.

Simple writing

You can do some manipulations on data and metadata as returned by `votable.load` (in lockstep) and write back the result using `votable.save(data, metadata, destF)`, where `destF` is a writable file object. This could look like this:

```
>>> data, metadata = votable.load("in.vot")
>>> tableDef = metadata.votTable.deepcopy()
>>> tableDef[V.FIELD(name="sum", datatype="double")]
>>> newData = [row+(row[1]+row[2],) for row in data]
>>> with open("res.vot", "w") as f:
>>>   votable.save(newData, tableDef, f)
```

Manipulating the table definitions is not convenient in this library so far. If you need such functionality, we would probably provide funtions to manipulation fields and params (or maybe just expose the child lists).

Iterative parsing

Iterative parsing is well suited for streaming applications and is attractive because the full document need never be completely in RAM at any time. For many applications, it is a little more clunky to work with, though.
The central function for iterative parsing is parse:

```
parse(inFile, watchset=[]) -> iterator
```

There also is `parseString` that takes a string literal instead of a file. `inFile` may be anything acceptable to the Elementtree library, so files and file names are ok.
The watchlist gives additional element types you want the iterator to return. These have to be classes from `votable.V` (see VOTable elements) By default, only special votable.Rows objects are returned. More on those below.
You get back stanxml elements with an additional attribute `idmap` is a dictionary-like object mapping ids to the elements that have so far been seen. This is the same object for all items returned, so forward references will eventually be resolved in this `idmap` if the input document is valid. For documents with clashing ids, the behaviour is undefined.
So, if you were interested in the resource structure of a VOTable, you could write:

```
>>> from gavo.votable import V
>>>
>>> for element in votable.parse(open("in.vot"), watchset=[V.RESOURCE]):
>>>   if isinstance(element, V.RESOURCE):
>>>     print "Starting new resource: %s"%element.name
>>>   else:
>>>     print "Discarding %s elements"%len(list(element))
```

Rows objects

Unless you order something else, parse will yield Rows objects only, one per TABLE element. Iterate over those to obtain the table rows, deserialized, with `None` as NULL. Note that you **must** exhaust the Rows iterator if you wish to continue parsing the table. You currently cannot simply skip a table.
The Rows' `tableDefinition` attribute contains the VOTable TABLE element that describes the current data.
To read the first table, do:

```
rows = votable.parse(open(inFileName)).next()
print rows.tableDefinition.iterChildrenOfType(votable.V.FIELD)
for row in rows:
  print row
```

The `tableDefinition` also lets you retrieve FIELDs by name using the `getFieldForName(name) -> FIELD` method.

stanxml elements

VOTables are built as stanxml elements; these have the common VOTable attributes as python attributes, i.e., to find the ucd of a FIELD `f` you would say `f.ucd`. To find out what's where, read the VOTable spec or check the VOTable elements and look out for class attributes starting with `_a_` -- these are turned into attributes without the prefix.
To access child elements, use any of the following methods:

- `iterChildren()` -- yields all the element's children in sequence
- `iterChildrenOfType(stanxmlType)` -- yields all the element's children that are an instance of `stanxmlType` (in a python inheritance sense)
- `makeChildDict()` -- returns a dictionary that maps child element names to sequences of children of this type.

So, to find a FIELD's description(s), you could say either:

```
list(f.iterChildrenOfType(V.DESCRIPTION))
```

or:

```
f.makeChildDict()["DESCRIPTION"]
```

The text content of a stanxml node is available in the `text_` attribute.

Post-data INFOs

More recent VOTable specifications allow INFO elements after table data. If you must catch these, things get a bit messier.
To be sure `tableDefinition` is complete including post-data groups, you need to let the iterator run once more after exhausting the data. Here's how to do this for the first table within a VOTable:

```
td = None
for rows in votable.parse(inFile):
  if td is not None:
    break
  td = rows.tableDefinition
  for row in rows:
    doMagic(row)
```

When you don't care about possible INFO elements anyway, use the simpler pattern above.

Generating VOTables

Low Level

When creating a VOTable using the low-level interface, you write the VOTable using DOM elements, using a simple notation gleaned from *Nevow Stan*. This looks like this:

```
vot = V.VOTABLE[
   V.INFO(name="generic", value="example")["This is an example"],
   V.RESOURCE[
     votable.DelayedTable(
       V.TABLE[
         V.FIELD(name="col1", datatype="char", arraysize="*"),],
       rows, V.BINARY)]]
```

-- square brackets thus denote element membership, round parentheses are used to set attributes.

The `votable.DelayedTable` class wraps a defined table and serializes data (`rows` in the example) using the structure defined by the first argument into a serialization defined by its last argument. Currently, you can use `V.BINARY` or `V.TABLEDATA` here. The data itself must come in sequences of python values compatible with your field definitions.

To write the actual VOTable, use the `votable.write(root, outputFile)` method. You can also call the root element's render() method to obtain the representation in a string.

High Level

There is a higher-level API to this library as a part of the DaCHS software. However, absent the mechanisms present there it's not trivial to come up with an interface that is both sufficiently expressive and simpler than just writing down stuff as in the low level API. Numpy arrays we'll do at some point, and it helps if you ask us.

Special behaviour

(a.k.a. "Bug and Features")

From the standards document it is not clear if, on parsing, nullvalue comparison should happen on literals or on parsed values. In this library, we went for literal comparison. This means that, e.g., for unsignedBytes with a null value of 0x10, a decimal 16 will not be rendered as None.

Values of VOTable type bits are always returned as integers, possibly very long ones.

Arraysize specifications are ignored when parsing VOTables in TABLEDATA encoding. The resulting lists will have the length given by the input. When writing, arraysizes are mostly enforced by clipping or padding with null values. They currently are not for strings and bit arrays.

One consequence of this is that with `arraysize="*"`, a NULL array will be an empty tag in TABLEDATA, but with `arraysize='n'` it will be n nullvalues.

Bit arrays in TABLEDATA encoding may have interspersed whitespace or not. When encoding, no whitespace is generated since this seems the intention of the spec.

All VOTables generated by this library are in UTF-8.

unicodeChar in BINARY encodes to and from UTF-16 rather than UCS-2 since UCS-2 is deprecated (and actually unsupported by python's codecs). However, this will fail for fixed-size strings containing characters outside of the BMP since it is impossible to know how many bytes an unknown string will occupy in UTF-16. So, characters for which

UCS-2 and UTF-16 are different will fail. These probably are rare, but we should figure out some way to handle this.

This will only make a difference for characters outside of the Basic Multilingual Plane. Hope you'll never encounter any.

Nullvalue declarations for booleans are always ignored. Nullvalue declarations for floats, doubles, and their complex counterparts are ignored when writing (i.e., we will always use NaN as a nullvalue; anything else would be highly doubtful anyway since float coming from representations in binary and decimal are tricky to compare at best).

When serializing bit fields in BINARY and there are too many bits for the number of *bytes* available, the most significant bits are cut off. If there are too few, zeroes are added on the left.

Post-data INFOs are not currently accessible when doing iterative parsing.

In BINARY serialization, fixed-length strings (both char and unicodeChar) are always padded right with blanks, whether or not a nullvalue is defined.

For char and unicodeChar arrays, nullvalues are supposed to refer to the entire array value. This is done since probably no library will support individual NULL characters (whatever that is) within strings, and this *if* we encounter such a thing, this probably is the meaning. Don't write something like that, though.

When deserializing variable multidimensional arrays from BINARY encoded streams, the length is assumed to be the total number of elements in the array rather than the number of rows. This may change when I find some VOTable using this in the wild.

Multidimensional arrays are returned as a single sequence on parsing, i.e. an arraysize of 5x7 is interpreted exactly like 35. This is not going to change. If you must, you can use the `unravel|Array(arraysize, seq)` function to reshape the list and get a nested structure of lists, where arraysize has the form of the VOTable FIELD attribute. If seq does not match the dimensions described by arraysize, the behavior is undefined (right now, we return short rows, but we may later raise exceptions).

On writing, you must flatten your multidimensional arrays before passing them to the library. This may change if people actually use it. The behavior then will be to accept as input whatever unravelArray returns. You can guess that the author considers multidimensional arrays a particularly gross misfeature within the misfeature VOTable arrays.

The `ref` attribute on TABLEs currently is not interpreted. Due to the way the library works, this means that such tables cannot be parsed.

The STC Data Model

To include STC information, you can just build the necessary GROUPs, PARAMs and FIELDrefs yourself.

Alternatively, you can install GAVO's STC library and build ASTs in some way (most likely from STC-S) and use the modelgroups module to include the information. This would look like this:

```
from votable import modelgroups, DelayedTable
[...]
ast = ast.parseQSTCS('Time TT "date" Position ICRS "ra" "de")
fields = [
  V.FIELD(name="date", datatype="char", arraysize="*"),
  V.FIELD(name="ra", datatype="float"),
  V.FIELD(name="de", datatype="float"),]
```

```
    table = V.TABLE[fields,
      modelgroups.marshal(ast, getIDFor)]
```

XXX TODO: Add id management.