pyCGNS.MAP/Manual

Release 4.2.0

Marc Poinot

October 24, 2012

CONTENTS

The MAP module is one of the most important modules of *pyCGNS*. MAP is the translator to get *CGNS/Python* trees from a *CGNS/HDF5* file and back, to save *CGNS/Python* trees as a *CGNS/HDF5* file. MAP uses *CHLone* and has no dependancy to *CGNS/MLL* or *CGNS/ADF*.

QUICK START

The MAPper is a module implementing the *CGNS/Python* mapping. There are only two functions in the module: the **load** and the **save**.

A simple exemple to load a *CGNS/HDF5* file as a *CGNS/Python* tree:

```python
import CGNS.MAP

(tree,links)=CGNS.MAP.load("T0.cgns")
```

The `tree` value contains the actual CGNS/Python tree, `links` value is a list of links found during the *HDF5* file parse. The default behavior of the load is to follow linked-to files. Now we can use the *CGNS.PAT* module to handle this tree. For example you can print the whole tree path hierarchy:

```python
import CGNS.MAP
import CGNS.PAT.cgnsutils as CGU

(tree,links)=CGNS.MAP.load("T0.cgns")

for p in CGU.getAllPaths(tree): print p
```

See the *CGNS.PAT* documentation on how to browse, modify such a GCNS/Python tree.

USER INTERFACE

MAP is a lightweight module, its purpose is to be as small as possible in order to be embedded separately in an application (not yet available). It uses the *CHLone* library and its Python binding. *MAP* is a powerful reader/writer of CGNS/HDF5 and CGNS/Python trees. The use of *CGNS.PAT* and the flags combination can help to achieve complex operations on a CGNS tree with a fast access and a low memory footprint. See the *examples*.

2.1 Functions

There are two functions: the `load` and the `save`. The `load` reads a *CGNS/HDF5* file and produces a *CGNS/Python* tree. The `save` takes a *CGNS/Python* tree and writes the contents in a *CGNS/HDF5* file:

```
(tree,links)=CGNS.MAP.load(filename,flags,depth,path,linkpaths,updatepaths)

status=CGNS.MAP.save(filename,tree,links,flags,depth,path)
```

The arguments and the return values are:

- **tree**: The `tree` is the list representing the CGNS/Python tree. The structure of a `tree` list is detailed in *SIDS-to-Python*. There is no link information in this tree either for *load* or for *save*.

  During the *load*, the links are silently replaced by the linked-to tree they are referring. The `links` value keeps track of these link references found while parsing the *CGNS/HDF5* file.

  During the *save*, the tree is splitted into separate files/nodes depending on the references found in the `links` value.

- **links**: The `links` is a list with the link node information. It is returned by a *load* and used as command parameters during the *save*. You can write your own `links` list or change the list you obtain after a *load*. The structure of a `links` list is detailed in *SIDS-to-Python*.

- **filename**: The name of the target file, to read or to write. The `filename` can be absolute or relative, it should be accessible in read/write depending on the action you perform on it. The file extension is unused.

- **flags**: You can control the behavior of a load/save using the *flags*. You have to look a these carefully, the same tree can load/save in a completely different way depending on these `flags`.

- **depth**: This positive integer value sets the level of children the load/save takes into account. For example, a depth of 2 would stop load/save the CGNS tree once the children of the children of the start node is reached. This level two child is used, its children are not. If you want to have all the children, use a 0 `depth` which means no limit on depth.

- **path**: The `path` defines the start node of the load/save. It should be an absolute path of an existing node in the argument filename. All the nodes along this path are taken into account for load/save actions.

- **linkpaths**: The load may need a *link files search path* if your linked-to files are not in the current directory. The `linkpath` argument is a list of strings, during the load *CGNS.MAP* will look for linked-to files using this list: it is parsed from the first element to the last, the selected file is the first found in this directory list.

- **updatepaths**: A dictionnary of paths (string) as keys and CGNS/Python nodes as values. When the load reaches a node with the path in the keys, the numpy value is updated instead of creating a new array. You can pass your own array with an already allocated memory zone or update and already loaded numpy.

> **Warning:** The current directory is **not** in the link search path. So if your linked-to file is in current directory, you should add . in the link search path list. If the filename is an absolute path name (not recommended !) then you should add and empty path in the search path list.

2.2 Flags

The flags are integers that can be OR-ed or XOR-ed to set/unset specific behavior during the load and the save. The boolean operators are used for the flag settings:

```
flags=CGNS.MAP.S2P_FOLLOWLINKS|CGNS.MAP.S2P_TRACE
```

```
flags =flags&~CGNS.MAP.S2P_TRACE
flags&=~CGNS.MAP.S2P_TRACE
```

The table below gives the *CGNS.MAP* flags.

Flag variable	*Function*
S2P_NONE	Clear all flags, set to zero.
S2P_ALL	Set all flags, set to one.
S2P_TRACE	Set the trace on, messages are sent to 'stdout'
S2P_FOLLOWLINKS	Continue to parse the linked-to tree (1)
S2P_MERGELINKS	Forget all link specifications. (2)
S2P_COMPRESS	Sets the compress flag for 'DataArray_t' (2)
S2P_NOOWNDATA	Forces the *numpy* flag \~NPY_OWNDATA (1) (3)
S2P_NODATA	Do not load large 'DataArray_t' (1)
S2P_UPDATE	Save updates existing file (6)
S2P_DELETEMISSING	not used

The S2P_DEFAULT flag corresponds to S2P_NONE | S2P_FOLLOWLINKS & S2P_REVERSEDIMS.

There is no requirements or check on which flag can or cannot be associated with another flag.

Remarks:

1. Only when you are *loading* a tree.

2. Only when you are *saving* a tree.

3. Which means all DataArray_t actual memory zones will **NOT** be released by Python.

3. The term *large* has to be defined. The *save* will **NOT** check if the CGNS/Python tree was performed with the S2P_NODATA flag on, then you have to check by yourself that your *save* will not overwrite an existing file with empty data!

4. The default behavior is to transpose array and dimensions of an array if this is not a NPY_FORTRAN array. If you set this flag to 1, no transpose would be performed and the array and its dimensions would be stored without modification even if the NY_FORTRAN flag is not there.

6. The file should exist, all new nodes are added, thus modifying the children list of their parents. Existing nodes are updated only in the case of value change. There no children removal, name or label change.

SIDS-TO-PYTHON MAPPING

3.1 CGNS/Python Tree

The CGNS/Python mapping defines a **tree** structure composed of **nodes** implemented for the *Python* programming langage. A special **links** structure is also defined for a correct mapping of the management of files on the disk. The mapping presented here is *NOT* a library [1], it is the lowest possible correspondance between a CGNS/SIDS structure and a Python representation. This specification is public and could be used as the basis for Python based CGNS application interoperability. *Python* is an interpreted langage and it has a textual representation of its objects, this representation can be used for CGNS/Python trees as well.

3.1.1 Commitment with CGNS standard

The mapping of the SIDS into a CGNS/Python structure uses the node as atomic structure. Comparing to CGNS/ADF or CGNS/HDF5, the contents of a node is unchanged in CGNS/Python. The way we represent data is different but all nodes attributes found in the section 6 of the *SIDS-to-ADF File Mapping Manual* [2] are applicable to the CGNS/Python mapping.

The **data type** mapping is changed compared to CGNS/ADF or CGNS/HDF5, the actual representation of basic types such as integers, floats and strings are closely mapped to the Python data types. See the table *Data types*.

Other elements of the node description are like the CGNS/ADF or CGNS/HDF5 mappings, in particular the **dimensions** and the order of these dimensions. The CGNS/SIDS section 3.1 states that the dimensions order should be the so-called *Fortran indexing convention* which states the column index is the first. The CGNS/Python nodes should respect this requirements.

> **Warning:** The Python arrays can be defined with either a *C* or a *Fortran flag*, this *flag* is used to set or to find the order used for the internal storage of an array. It has **no effect** on the dimensions of a *numpy* array, but on its internal memory layout. It's up to the user to manage this flag and its impact on the use of an array, in particular for the read/write on the disks through the C API.

For example, section 6.1.2.2 describes the `DimensionalUnits_t` node with dimensions values `(32,5)`. This should be understood as *Fortran* order values, and thus `(32,5)` should be found as this in the *shape* of the *numpy* array [3] whichever status the *Fortran* flag set has.

A *numpy* array with the *C* flag set should also have a shape of `(32,5)`, again, the internal representation of this *C* array has to be taken into account during read/write operations.

See the *C API* and *Examples and Tips* sections about this requirement and its impact on *numpy* array use.

---

[1] The Open Source *pyCGNS* Python module defines services on the top of this CGNS/Python mapping.

[2] The *SIDS-to-ADF* [CG2] or *SIDS-toHDF5* [CG3] documents of *cgns.org* have the detailed description of each node of the standard.

[3] We show in the textual representation section that this dimension ordering could lead to quite complicated Python code, but our choice was to take the implementation from the 'Fortran' world, which is the basis of the CFD world. It would be up to the user to write his own application layer for a better Python interface.

3.1.2 The node structure

The structure of a CGNS data set is held in a so-called **CGNS/Python tree**. The tree is composed of nodes, each node may have children which are nodes too. The node structure is a python sequence (i.e. list or tuple), composed of four entries: the name, the value, the list of children and the type.

Attribute	*type*
Name	string
Value	*numpy* array
Children	list of CGNS/Python nodes
Type	string

The CGNS/Python mapping requires that:

The *name* is a Python string, it should not be empty. The name should not have more than 32 chars and should not have / in it. The names `.` (a single dot) And `..` (dot dot) are forbidden (but names with dot and something else are allowed, for example `Zone.001` is ok).

The representation of *values* uses the numpy library array. It makes it possible to share an already existing memory zone with the Python object. The numpy mapping of the values is detailed hereafter. An empty value should be represented as None, any other value is forbidden.

The *children list* can be a list or a tuple. The use of a list is strongly recommended but not mandatory. A read-only tree can be declared as a tuple. It is the responsibility of the applications to parse sequences wether these are lists or tuples. A node without child has the empty list `[]` as children list.

The *type* is the Python string representation of the CGNS/SIDS type [4] (i.e. it is the same for CGNS/ADF or CGNS/HDF5). A type string cannot be empty.

We have now a typical CGNS/Python node, which can be represented with the pattern [5]:

```
node = [ <name:string>, <value:numpy.array>, [ <child:node>* ], <cgns-type:string> ]
```

We use there the textual representation of a Python object. All the Python types used in this CGNS/Python mapping have a full textual representation. This is detailed in the next section.

The order of the values is significant, for example `node[0]` should always be the name of the node (Python has an index ordering starting with zero)

We see now that a CGNS/Python tree is a node. This node has children which have children and so on... Any node can be held as a subpart of a complete tree, we say each node is a *sub-tree*. Our CGNS/Python tree has a *root* node which is its first node. There is no clear definition of a *root* node in the CGNS/SIDS or in the SIDS mappings.

In the case of a *CGNSBase_t* level node, the CGNS/ADF or CGNS/HDF5 defines a sound node which can be mapped to CGNS/Python. However, the CGNS/SIDS states that several bases can be found in a CGNS tree. The father node of a base would have the pattern:

```
root = [ <CGNSLibraryVersion:node>, <CGNSBase:node>* ]
```

Which is not consistent with a *normal* node. We want to remove this exception, we define a CGNS/Python tree root, or first node, as a list with a compliant CGNS/Python node. which is not the *node* pattern. Then the applications have to have a specific way to manage this first node. This lack of *root* node is not that important when you use the CGNS/MLL because the function are hidding the actual node implementation. With CGNS/Python, the user can manage the nodes as true Python objects, and we have to provide him with a sound interface, or at least as sound as possible. For this consistency reason, the CGNS/Python mapping defines a new type for the *root* node, see the *CGNSTree_t type* section.

---

[4] The CGNS/SIDS type (see [CG1]) is the type of the node, *NOT* the type of the data contained into the node.

[5] The syntax is: `<A:T>` with A attribute name and T attribute type. The types are detailed in another section. The `<A:T>*` means zero or more `<A:T>` separated by `,` if more than one.

3.1.3 Textual representation

It is possible to declare a CGNS/Python node as a textual representation. There is a exemple of a zone connectivity sub-tree with the CGNS/Python in textual mode, a simple `PointRange` node with two 3D indices:

```
pr=['PointRange',
    numpy.array([[1,25],[1,9],[1,1]],dtype=numpy.int32,order='Fortran'),
    [],
    'IndexRange_t']
```

The `PointRange` node has no child, the children list is an empty list. The values of the array are initialized with a list, the order of the elements in the list matches the *Fortran* indexing: in that example the first point indices are `[1,1,1]` and the second point indicies are `[25,9,1]`.

The evaluation of this string by the Python interpreter creates a CGNS/Python compliant node as a Python list. Please note the types of this `pr` node, there are only native Python types (list, string, integer) and *numpy* types or enumerates. You have to have a variable to hold the node or the CGNS sub-tree, if you have no reference to the actually created Python objects these will be unreachable and thus garbaged.

The textual representation can be *import*-ed as any Python textual file, with all possible Python use you can imagine.

> **Warning:** The Python lists are objects. When you refer to a list you do not copy this list unless you ask for such a copy. This is important because if you modify an existing list you modify an object that could be used by others. In the CGNS/Python mapping the children of a node is a list of nodes. If you refer to such a list without a copy, any modification of this child list will impact nodes using this list. This is detailled in the section *Examples and Tips* .

3.1.4 Numpy array mapping

A CGNS/Python node value is a *numpy* array, this python object contains the **number of dimensions**, the **dimensions**, the **data type** and the actual **data** array. Then this implicit information is not a part of the *node* structure. As we really want to have the most generic node as possible, we require that even single dimension values should be stored as *numpy* array. A single integer, float or a single string should be embedded into a *numpy* array.

As we mentionned before, an empty value has to be represented by `None` which is a native Python value, not a *numpy* value:

```
gc=['Grid#002',None,[cx,cy,cz],'GridCoordinates_t']
```

Here `cx`, `cy`, `cz`, are nodes, not arrays.

The *numpy* end-user interface makes it possible to define some of these required data as deduction of required parameters. The number of dimensions is the size of the so-called shape. The dimensions can be forced for empty values or can be deduced from the data itself:

```
a=numpy.array([1.4])
b=numpy.ones((5,7,3),'i')
```

The first declaration has dimension 1, number of dims 1, data type `float64`, all deduced from the data declaration, the second has dimensions `(5,3,7)`, number of dimensions 3, data type set as `int32`.

A *numpy* array can be declared as *C order* or *Fortran order*. There is no requirements in this mapping wether the internal layout of the memory should be *C* or *Fortran*. However, an array should have a shape with the same order of dimensions as described in the *SIDS-to-ADF File Mapping Manual* ([CG2]).

> **Warning:** If you use the Python C API, it is the responsability to the application to check the *numpy* ordering flag and to manage the arrays with respect to memory layout. See the *C API* section.

The way to get the node data information regarding the [CG2] datatypes and dimensions requirements is to access to the *numpy* object attributes:

```
pr=numpy.array([[1,2,3],[4,5,6]])

dims=pr.shape
ndims=len(pr.shape)
datatype=pr.dtype
fortranorder=numpy.isfortran(pr)
corder=not numpy.isfortran(pr)
```

3.1.5 Data types

A value is a numpy array, the contents of an array is homogeneous and has a data type. The data types of your CGNS/Python arrays depends on the data type as defined in [CG2].

The type of the data can be set at the creation time, the *numpy* type is associated to the *ADF* type required by the CGNS/SIDS. A bad data type, even if it silently looks like the result you want, would lead to an non-compliant CGNS tree. The required mapping for the end-user interface uses the types :

ADF type	*Numpy type(s)*	*Remarks*
I4	'i' int32	(1)
I8	'l' int64	(2)
R4	'f' float32	(3)
R8	'd' float64	(4)
C1	'c' '|S1'	(5)

All other *ADF* or *numpy* types are ignored. The string type is a bit special, see the remark (5) about the strings used in *numpy* arrays.

Remarks:

1. The 32bits precision has to be forced, the default integer size in python the `int64` data type. To create an *I4* array, you can use:

   ```
   numpy.array([1,2,3],'i',order='Fortran')
   ```

2. The 64bits precision is the default integer in python. To create an *I8* array, you can use:

   ```
   numpy.array([1,2,3],order='Fortran')
   ```

3. The 32bits precision has to be forced, the default float size in python is `float64`. To create an *R4* array, you can use:

   ```
   numpy.array([1.4],'f',order='Fortran')
   ```

4. The 64bits precision is the default float in python. To create an *R8* array, you can use:

   ```
   numpy.array([1.4],order='Fortran')
   ```

5. The array has to be created as a char multi-dimensionnal array. An incorrect creation with a simple statement such as: `numpy.array('GoverningEquations')` produces a *wrong* zero dimension array. The correct creation for a single value could be: `numpy.array(tuple('GoverningEquations'),'|S1')` where the shape (i.e. the dimensions of the array) is `(18,)`.

3.2 Specific CGNS/Python topics

3.2.1 The CGNSTree_t type

The tree structure of a CGNS data set is broken by the exception of the root node. We take the opportunity of this new CGNS/Python mapping to add a consistent root node for the CGNS tree [6].

The *CGNSTree_t* type is a node with the pattern:

```
root= [ <name:string>, None, [ <CGNSLibraryVersion:node>, <CGNSBase:node>* ], 'CGNSTree_t' ]
```

The children list is the CGNS/ADF-like root node. The *CGNSTree* node has a user-defined name, no value and a fixed *CGNSTree_t* type.

3.2.2 Legacy CGNS types alternative

The CGNS/SIDS defines all CGNS types and has a rule to suffix them with `_t`. There are some exceptions where some CGNS/SIDS types have been translated into strings with a special syntax.

The CGNS/Python mapping allows the use of alternate types for these, the user can either use the legacy type or the alternate CGNS/Python type. The alternate types are:

CGNS/SIDS type	CGNS/Python optional type
`"int[1+...+IndexDimension]"`	`DiffusionModel_t`
`"int[IndexDimension]"`	`Transform_t`
`"int[IndexDimension]"`	`InwardNormalIndex_t`
`"int"`	`EquationDimension_t`

Please note the `["]` character which is part of the CGNS legacy type.

> **Warning:** This CGNS/Python feature adds *NON-SIDS* type(s) and this should be added or removed by the user application during the read and the write to the disk with a CGNS/ADF or CGNS/HDF5 compliance. The CGNS.MAP module has an option to check and remove these alternate types. As long as your application has interoperability with another CGNS/Python application there should be no problem.

3.2.3 Links

The **links** are used to set and get CGNS symbolic links information. This information is relevant only during read/write operations on disks. A CGNS/Python tree cannot have embedded links, as this tree is a list of lists making a link to another list is non-sense in Python [7]. The **links** list is an extra information, not embedded into the CGNS/Python tree, and only used as disk-related operations.

> **Warning:** In the case a CGNS/Python application would not like to follow a link and then to have some *missing* data in its CGNS tree, the so-called *linked-from* node has to be removed from its parent children list.

This **links** list is an unsorted list of *link-entries* with only one entry per link. A *link-entry* is an ordered list of Python string values:

> The *target directory name* is the linked-to directory name, as it would be used to open it. It should be a valid absolute/relative file path as a plain Python string or `None`.

> The *target file name* is the linked-to file name, as it would be used to open it. It should be a valid absolute/relative path as a plain Python string. Its path-prefix part and its file extension part can be empty but the filename itself cannot.

---

[6] This CGNS/Python feature adds *NON-SIDS* type(s) and this should be added or removed by the user application during the read and the write to the disk with a CGNS/ADF or CGNS/HDF5 compliance.

[7] A Python list is a reference, if you put a list as a child of another list the Python interpreter actually refers to the child list. Then a child can be shared by two different lists if you do not ask for a copy. In other words, the links are the natural way of referencing to lists in Python.

The *target node name* is the linked-to node name as a plain Python string. It should be the **absolute** path of the node in the linked-to file. This value cannot be empty.

The *local node name* is the **absolute path** of the node in the source Python/CGNS tree. This plain Python string cannot be empty.

The links with a second level file, in other words the links in a file you are parsing after following a first link, are **always** referred as if you where in the *target filename*. Then, a list of links can be reused from one parse to another, because the `links` list is relative to the target file. The example hereafter can be an *input* as well as an *output* links list, an application would set it for a *save* or get it from a *load*:

```
[ ['/tools/CFD/ref#M6','M6_A.cgns'  ,'/Base#1/ReferenceState',
                                     '/Base/ReferenceState'],
  ['/tmp/restart'     ,'M6#001.cgns','/Base#1/Zone1/FlowSolution#EndOfRun',
                                     '/Base/Zone1/FlowSolution#Init']
]
```

The *target directory name* information is distinct to the filename, because you can have different actual target files depending on the search paths you set. This information is relevant as ouput from the read of an actual file, it should be set to *None* or ignored for a write. During a write, the only information taken into account should be the *target file name*, *target node name* and the *local node name*.

In the example above, the entries are interpreted in a different way depending if they are result of a *read* or directives for a *write*. In the case of a read, the first entry means that the file we have read has a node /Base/ReferenceState which is a link to the node /Base#1/ReferenceState in the file M6_A.cgns. The first directory of the file search path in which the file M6_A.cgns has been found is /tools/CFD/ref#M6. In the case of a *write*, the same entry means that the application should create a link for the node /Base/ReferenceState when it reaches it. This link would have M6_A.cgns as target file and /Base#1/ReferenceState as target node. The /tools/CFD/ref#M6 value is ignored.

> **Warning:** The links list is relative to the current tree. If you want to track links of links your application has to manage this by itself, setting or getting links list during the different tree traversals.

3.2.4 C API

There is no requirement on the way you would create or manage a numpy array at the C API level. But you have to remember that the definition of the node contents is SIDS-to-ADF which states that data arrays and index ordering use the *Fortran* convention.

You can manage all your numpy arrays with the C order in memory, but you have to be sure that the storage on the disk, i.e. using ADF or HDF5, has the correct fortran orders. The storage also has to be contiguous in the memory. When you create or obtain a copy of a numpy array you can set a flag to force a C or Fortran ordering: one of the NPY_CCONTIGUOUS or NPY_FCONTIGUOUS flag can be set. In the case of a NPY_CCONTIGUOUS flag set, it is up to the application to set a Fortran memory layout and a Fortran index ordering while reading/writing data to/form a CGNS/ADF or CGNS/HDF5 file [8].

The *numpy C API* allows the share of memory zone. In other words you can have a *Fortran* or *C* array you can directly set as your *numpy* array without duplication. You can reduce the memory use when your application can handle this, you can also set the NPY_OWNDATA flag to indicate to *numpy* that it should not release the array memory when the *numpy* array object is garbaged.

3.3 Examples and tips

Python comes from the C world, as well as the numpy library. This means that many behavior are assuming C-order in dimensions. The CGNS/Python mapping states that arrays should have a Fortran indexing for their actual data and that the dimension order of the data is those detailed in the [CG2] and [CG3] documents.

---

[8] For example, *CGNS.MAP* detects th NPY_FCONTIGUOUS and forces a data and dimensions transpose during the read/write (unless the user forces the *CGNS.MAP.S2P_NOTRANSPOSE* flag in the `load` or the `save`).

We give here some known issues and tips to handle this Fortran indexing in CGNS/Python. We use specific CGNS/SIDS structures to illustrate our examples.

3.3.1 IndexRange_t

The *IndexRange_t* is an integer array of dimensions *(IndexDimensions,2)* as detailed in [1]. The node data, in the example here, is two points with three indices. The *Python-ish* way to define them is to have a list of two lists of integers, which leads to problems if you forget your fortran order. We want to set a node with the following Python code:

```
node=['PointRange', a, [], 'IndexRange_t']
```

Now we see how to declare a correct *a* variable as a *numpy* array. If you do not specify an order to *numpy*, the default is the C-order:

```
>>> a=numpy.array([[1,2,3],[4,5,6]],dtype=numpy.int32)
>>> numpy.isfortran(a)
False
>>> a[0]
array([1,2,3], dtype=int32)
>>> a.shape
(2,3)
```

This *numpy* array is correct but you would have to transpose dimensions are memory layout before a storage on disk. Or you can enter the list itself using an explicit Fortran-order:

```
>>> a=numpy.array([[1,4],[2,5],[3,6]],dtype=numpy.int32)
>>> numpy.isfortran(a)
False
>>> a[0]
array([1,4], dtype=int32)
>>> a.shape
(3,2)
```

In that case, the *shape* is correct but the user has no mean to know wether your convention is C or Fortran. You can set the fortran flag for this. The possible creation of the array above is then:

```
>>> a=numpy.array([[1,4],[2,5],[3,6]],dtype=numpy.int32,order='Fortran')
>>> numpy.isfortran(a)
True
>>> a[0]
array([1,4], dtype=int32)
>>> a.shape
(3,2)
```

Then an application can detect your array has *Fortran* order and should be stored as found without any transpose.

3.3.2 IndexArray_t

There is another example switching from one order to another, this is used to add a point in a list in an easier way

```
node=['PointList', a, [], 'IndexArray_t']
```

The possible creation of the array `a` above is then:

```
>>> a=numpy.array([[1,4],[2,5],[3,6]],dtype=numpy.int32,order='Fortran')
>>> a
array([[1, 4],
       [2, 5],
       [3, 6]], dtype=int32)

>>> a=numpy.array(a.T.tolist()+[[7,8,9]],dtype=numpy.int32,order='Fortran').T
```

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]], dtype=int32)
```

You see that the syntax is completely unreadable, we use the *numpy* transpose attribute `T` to switch from *Fortran* to *C* order and back.. If you start with the *C* order, the Python syntax is clear:

```
>>> a=numpy.array([[1,2,3],[4,5,6]],dtype=numpy.int32)
>>> a
array([[1,2,3],
       [4,5,6],  dtype=int32)
```

```
>>> a=numpy.array(a.tolist()+[[7,8,9]],dtype=numpy.int32)
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]], dtype=int32)
```

And the application in charge of the *write* to the disk that would detect the abscence of *Fortran* flag and then transpose the array and its dimension.

3.3.3 DimensionalUnits_t

This node contains strings. The strings are an issue in CGNS/Python because we want to use the raw level for *numpy* (instead of *numpy* module proposed for string manipulation). We want to keep a common interface for all nodes and we do not want an exception with strings. The *DimensionalUnits_t* node can be defined as:

```
node=['DimensionalUnits', a, [], 'DimensionalUnits_t']
```

Now we see how we can defined the *numpy* array in variable *a*. The *DimensionalUnits_t* states we need a *(32,5)* array of chars. In the case of a fixed size multi-dimensionnal string array, each entry should be split as a sequence with a fixed max size (usually 32 chars):

```
a=numpy.array([
            tuple('%-32s'%'Kilogram'),
            tuple('%-32s'%'Meter'),
            tuple('%-32s'%'Second',)
            tuple('%-32s'%'Kelvin'),
            tuple('%-32s'%'Radian'),
          ],'|S32',order='Fortran').T
```

The shape of the resulting array is `(32,5)` again note the `T` at the end of the command which produces the transpose. You can use a `S32`, `|S1` or `c` type directive. An important point in this *string* as an array is the trailing *spaces* you have to fill the array cell. You have to use a `string.strip` before any string operation unless your Python application is aware of this *forced* size.

3.3.4 Zone_t

There we have an interesting example with the use of a data of a node. The *Zone_t* node has the dimensions of the *zone*. These dimensions are a data and theses data values should be used as *dimension* attribute of the children nodes. In other words, the user takes the *Zone_t* dimensions and creates a *numpy* array with them:

```
zonenode=['Zone001',zonedims,zonechildrenlist,'Zone_t']
```

The `zonedims` *numpy* array can b set as:

```
zonedims=numpy.array([[3,2,0],[5,4,0],[7,6,0]],dtype=numpy.int32,order='Fortran')
```

in the case of a 3D structured zone with `(ni,nj,nk)=(3,5,7)`. If you want to create a solution array with these dimensions, you can to use the following syntax:

```
zonevertexsize=zonedims[:,0]
zonecellsize=zonedims[:,1]
zonevertexboundarysize=zonedims[:,2]
```

This numpy syntax allows the user to take the whole column as a so-called *slice*.

3.3.5 Sub-tree imports

For example, the following snippet *imports* a truncated *ReferenceState*:

```python
import numpy

refvalues=[
            ['Mach',numpy.array([0.2]),[],'DataArray_t']
            ['Reynolds',numpy.array([23300000.0]),[],'DataArray_t']
            ['LengthReference',numpy.array([0.5]),[],'DataArray_t']
            ['Density',numpy.array([1.22524863848]),[],'DataArray_t']
]

data=['ReferenceState',None,refvalues,'ReferenceState_t']
```

Once *import*-ed, your Python code can insert this node in its structure (here our previous code snippet is in the file `refstate.py`:

```python
import numpy
import refstate

tree=['CGNSTree',None,[],'CGNSTree_t']
base=['Fuselage',numpy.array([3,3],dtype=numpy.int32),[],'CGNSBase_t']

tree[2].append(base)
base[2].append(refstate.data)
```

3.3.6 Sub-tree share

A list is a reference. If you put a list into another one, you do not perform a copy, you use a reference. Then the modification of the first list is in the second:

```python
>>> a=[1,2,3]
>>> b=[a,[4,5,6]]
>>> b
[[1, 2, 3], [4, 5, 6]]

>>> a[1]=9
>>> b
[[1, 9, 3], [4, 5, 6]]
```

You always have to take care of the lists, in particular if you use large CGNS/Python trees you want to share to optimize memory. Another point to keep in mind is that *numpy* copies do *NOT* propagate *Fortran* flag.

EXAMPLES

4.1 MAP Examples

The *CGNS.MAP* provides the user with the two functions `load` and `save`. The actual actions these functions are performing heavily depends on the arguments you pass to these functions, the `load` takes a CGNS/HDF5 tree and returns at least a CGNS/Python tree, while the `save` takes a CGNS/Python tree and writes (modifies) a CGNS/HDF5 tree. Rather than listing all options, we suggest your try to figure out how to use these functions reading the examples.

All the examples are assuming the following imports:

```python
import CGNS.MAP
import CGNS.PAT.cgnsutils    as CGU
import CGNS.PAT.cgnskeywords as CGK
import CGNS.PAT.cgnslib      as CGL

import numpy as NPY
```

4.1.1 Complete load of a *CGNS/HDF* file

The translation of a CNS/HDF5 file into a CGNS/Python tree is performed with the simple line:

```python
(tree,links)=CGNS.MAP.load("testfile.cgns")

(tree,links)=CGNS.MAP.load("001Disk.hdf",CGNS.MAP.S2P_DEFAULT|)

print tree
```

To be more precise, you can add the *depth*, the *path* and the *linkpath* of the file. In our case, the *depth* is worth 0 and the string's *path* is '' because we want to load entirely the tree. Here, the *linkpath* is an empty list because this file has'nt *links*. As we are located in the same directory as the file, we don't owe necessarily write the *path*.

```python
import CGNS.MAP

(tree,links)=CGNS.MAP.load("001Disk.hdf", CGNS.MAP.S2P_DEFAULT,0, '', [], None)

print tree
```

4.1.2 Complete save of a *CGNS/Python* tree

To save a *CGNS/Python* tree, you have to use the *save* function which takes a CGNS/Python tree and writes the contents in a CGNS/HDF5 file. As the previous case, it's necessary to specify the file's name and its flags.

```
import CGNS.MAP

status=CGNS.MAP.save("filename.hdf", tree, links, CGNS.MAP.S2P_DEFAULT)
```

4.1.3 Load of a *CGNS/HDF* file skeletton

You want to parse a GCNS/Python tree without the actual data (numpy arrays). This is usefull when you want to find out specific nodes before actually load them:

```
flags=CGNS.MAP.S2P_DEFAULT|CGNS.MAP.S2P_NODATA
(tree,lk)=CGNS.MAP.load("data/T0.cgns",flags,0,None,[],None)

# find all zones
zlist=CGU.getAllNodesByTypeList2([CGK.Zone_ts],tree)
```

In this example, we can also set the *depth* to 3 instead of 0. Then the parse would stop at the zone level. If you set *depth* to 0 there is no depth limit, a level of 1 means only *CGNSTree_t*, a level of 2 means *CGNSLibraryVersion_t*, *CGNSBase_t* and so on...

In order to load the skeleton of a *CGNS/HDF* file, we use the *load* function with the *S2P_NODATA* flag. This flag acts by removing all the data contained in the tree.

```
import CGNS.MAP

flags=CGNS.MAP.S2P_DEFAULT|CGNS.MAP.S2P_NODATA

(tree,link)=CGNS.MAP.load("001Disk.hdf", flags, 0, '', [], None)

print tree
```

4.1.4 Update a sub-tree

An existing CGNS/Python tree is read again from a file. The *update* dictionnary declares the nodes you want to update instead of creating new ones:

```
updict={}
path='/Disk/zone1/GridCoordinates/CoordinateZ'
updict[path]=CGU.getValueByPath(tree,path)
path='/Disk/zone2/GridCoordinates/CoordinateX'
updict[path]=CGU.getValueByPath(tree,path)

(tree,lk)=CGNS.MAP.load("data/T0.cgns",CGNS.MAP.S2P_DEFAULT,0,None,[],updict)
```

Then, we modify the data contained in the node whose the label is 'fluid' by replacing 'ideal' instead of 'perfect'.

```
>>> tree[2][1][2][1][2][5]
['fluid',array(['I','d','e','a','l'],dtype='|S1'),[],'DataArray_t']

>>> import numpy
>>> tree[2][1][2][1][2][5][1]=numpy.array(tuple('parfait'),'|S1')

>>> tree[2][1][2][1][2][5]
['fluid',array(['p','a','r','f','a','i','t'],dtype='|S1'),[],'DataArray_t']
```

4.1.5 Copy a part of a tree

To copy only a part of a tree, you have to use the *load* function by mentioning the *path* which defines the start node of the load. In our example, we load only the node whose label is 'fluid' because this node has no children and the nodes '.SOLVER#Compute' and 'Disk' which are parents of the node 'fluid'.

import CGNS.MAP

path='/Disk/.Solver#Compute/fluid' (tree,links)=CGNS.MAP.load("001Disk.hdf",CGNS.MAP.S2P_DEFAULT,0,path,[],Non

```
>>> print tree
```

['CGNSTree', None, [['Disk', array([3, 3], dtype=int32), [['.Solver#Compute', None, [['fluid', ar-ray(['I', 'd', 'e', 'a', 'l'], dtype='|S1'), [], 'DataArray_t']], 'UserDefinedData_t']], 'CGNSBase_t']], 'CGNSTree_t']

4.1.6 Copy and merge links

In order to extract the *links*, you have to specify the path where are situated the other files which are linked to our file in a list. For example, we are looking for the link files which are in the directory 'liens'.

4.1.7 Some usual issues

Your load leads to an empty tree:

```
>>>import CGNS.MAP
>>>(tree,links)=CGNS.MAP.load("./T0.cgns",CGNS.MAP.S2P_FOLLOWLINKS)
>>>tree
['CGNSTree', None, None, 'CGNSTree_t']

import CGNS.MAP
```

your file is an ADF file, not an HDF5 file. (tree, links)=CGNS.MAP.load("./liens/3element_airfoll_msh_cfd.hdf", CGNS.MAP.S2P_DEFAULT,0,'`,['./liens/'],None)

print links

GLOSSARY

cgns.org The official CGNS web site, by extension any document on this web site has an *official* taste...

CGNS The specific purpose of the CFD General Notation System (CGNS) project is to provide a standard for recording and recovering computer data associated with the numerical solution of the equations of fluid dynamics. See also the *How to?*.

CGNS/SIDS The Standard Interface Data Structure is the specification of the data model. This public document describes the syntax and the semantics of all tree-structured data required or proposed for a CFD simulation.

CGNS/MLL The Mid-Level Library is an example implementation of *CGNS/SIDS* on top of *CGNS/ADF* and *CGNS/HDF5* mappings. This library has a C and a Fortran API.

CGNS/ADF The Advanced Data Format *CGNS/SIDS* implementation. A binary storage format and its companion library, developped by *Boeing*.

CGNS/HDF5 The Hierarchical Data Format *CGNS/SIDS* implementation. A binary storage format and its companion library (see below).

CGNS/Python The Python programming language *CGNS/SIDS* implementation.

CHLone A *CGNS/HDF5* compliant implementation. The CHLone library is available on SourceForge.

HDF5 A powerful storage system for large data. The HDF5 library should be seen as a middleware system with a lot of powerful features related to efficient, portable and trustable storage mean.

python An object oriented interpreted programming language.

cython A compiler tool that translate Python/Numpy into C code for performance purpose.

numpy The numerical library for Python. *Numpy* is used to store the data in Python arrays which have a direct memory mapping to actual C or Fortran memory.

VTK A visualization toolkit used to display 3D objects ni *CGNS.NAV*.

PySide The Python interface for the Qt toolkit. PySide

Qt A powerful graphical toolkit available under GPL v3, LGPL v2 and a commercial license. The current use of Qt is under LGPL v2 in pyCGNS.

5.1 MAP Index

- *genindex*

BIBLIOGRAPHY

[CG1] CGNS SIDS - Standard Interface Data Structure http://www.grc.nasa.gov/WWW/cgns/sids

[CG2] SIDS-to-ADF Mapping Reference Manual http://www.grc.nasa.gov/WWW/cgns/filemap

[CG3] SIDS-to-HDF Mapping Reference Manual http://www.grc.nasa.gov/WWW/cgns/filemap_hdf

[PY1] Python Programming langage http://www.python.org

[PY2] Numpy http://numpy.scipy.org