



# Internal Documentation

## *Release 3.0*

**/ELSA/MU-09020/V3.0**

**Dec 05, 2019**



# CONTENTS

<b>1</b>	<b>Preamble</b>	<b>1</b>
<b>2</b>	<b>List of functions</b>	<b>3</b>
<b>3</b>	<b>Contents</b>	<b>11</b>
3.1	Node tests . . . . .	11
3.2	Set/create generic nodes . . . . .	15
3.3	Access nodes . . . . .	20
3.4	Check nodes . . . . .	40
3.5	Copy nodes . . . . .	43
3.6	Add/remove node . . . . .	46
3.7	Modify nodes . . . . .	51
3.8	Create specific CGNS nodes . . . . .	55
<b>4</b>	<b>Index</b>	<b>97</b>



## PREAMBLE

This module provides simple and efficient functions for creating/traversing/manipulating CGNS/Python data tree (referred here as *pyTree*).

This module is part of Cassiopee, a free open-source pre- and post-processor for CFD simulations.

To use the module:

```
import Converter.Internal as Internal
```

All functions of Cassiopee modules (`Converter.PyTree`, `Transform.PyTree`, ...) work on 3 types of containers: a container for grid coordinates named `__GridCoordinates__`, a container for node solution named `__FlowSolutionNodes__` and a container for center solution named `__FlowSolutionCenters__`. By default:

```
Internal.__GridCoordinates__ = 'GridCoordinates'  
Internal.__FlowSolutionNodes__ = 'FlowSolution'  
Internal.__FlowSolutionCenters__ = 'FlowSolution#Centers'
```

To make the functions operate on another named container for example 'FlowSolution#Init', start your python script with:

```
Internal.__FlowSolutionNodes__ = 'FlowSolution#Init'
```

To automatically set the container names used by Cassiopee to the ones that exists in your `pyTree t`, use:

```
Internal.autoSetContainers(t)
```

If multiple containers exists in your `pyTree`, for instance 'FlowSolution1' and 'FlowSolution2' for 'FlowSolution\_t', the the first met is used by Cassiopee.

Through all this documentation, a `pyTree` node is a python list: **['NodeName', value numpy array, [node children list], 'Type\_t']** as described by CGNS/python standard ([https://cgns.github.io/CGNS\\_docs\\_current/python/sidstopython.pdf](https://cgns.github.io/CGNS_docs_current/python/sidstopython.pdf)).

*'NodeName'* is a string describing the node **name**; *value numpy array* is a numpy array corresponding to the **value** stored in node; *'Type\_t'* is a string describing the node **type**; *node children list* is a list of pyTree nodes that are the children of this node.

## LIST OF FUNCTIONS

### – Node tests

<code>Converter.Internal.isTopTree(node)</code>	Return True if node corresponds to a top tree node (CGSNTree_t).
<code>Converter.Internal.isStdNode(node)</code>	Return 0 if node is a list of standard pyTree nodes, -1 if node is a standard pyTree node, -2 otherwise.
<code>Converter.Internal.typeOfNode(node)</code>	Return the type of node as an integer.
<code>Converter.Internal.isType(node, ntype)</code>	Return True if node is of given type.
<code>Converter.Internal.isName(node, name)</code>	Return True if node has given name.
<code>Converter.Internal.isValue(node, value)</code>	Return True if node has given value.
<code>Converter.Internal.isChild(start, node)</code>	Return True if node is a child of node start.

### – Set/create generic nodes

<code>Converter.Internal.setName(node, name)</code>	Set name in node.
<code>Converter.Internal.setType(node, ntype)</code>	Set type in node.
<code>Converter.Internal.setValue(node[, value])</code>	Set given value in node.
<code>Converter.Internal.createNode(name, ntype[, ...])</code>	Create a pyTree node.
<code>Converter.Internal.addChild(node, child[, pos])</code>	Add a child node to node's children.
<code>Converter.Internal.createChild(node, name, ntype)</code>	Create a node and attach it to node's children.
<code>Converter.Internal.createUniqueChild(node, ...)</code>	Create a unique child in node's children.

### – Access nodes

<code>Converter.Internal.getName(node)</code>	Return node name.
<code>Converter.Internal.getType(node)</code>	Return node type.
<code>Converter.Internal.getChildren(node)</code>	Return children list of a node.
<code>Converter.Internal.getValue(node)</code>	Return the value of a node.
<code>Converter.Internal.getPath(t, node[, py-CGNSLike])</code>	Return the path of node.
<code>Converter.Internal.getNodesFromName(t, name)</code>	Return a list of nodes matching given name.
<code>Converter.Internal.getNodeFromName(t, name)</code>	Return the first matching node with given name.
<code>Converter.Internal.getByName(t, name[, ...])</code>	Return a standard node containing the list of matching name nodes as children.
<code>Converter.Internal.getNodesFromType(t, ntype)</code>	Return a list of nodes matching given type.
<code>Converter.Internal.getNodeFromType(t, ntype)</code>	Return the first matching node with given type.
<code>Converter.Internal.getByType(t, ntype[, ...])</code>	Return a standard node containing the list of matching type nodes as children.
<code>Converter.Internal.getNodesFromNameAndType(t, ...)</code>	Return a list of nodes matching given name and type.
<code>Converter.Internal.getNodeFromNameAndType(t, ...)</code>	Return the first matching node with given name and type.
<code>Converter.Internal.getNodesFromValue(t, value)</code>	Return a list of nodes matching given value.
<code>Converter.Internal.getParentOfNode(t, node)</code>	Return the parent of given node.
<code>Converter.Internal.getParentFromType(start, ...)</code>	Return the first parent node matching type.
<code>Converter.Internal.getParentsFromType(start, ...)</code>	Return all parent nodes matching type.
<code>Converter.Internal.getNodePosition(node, parent)</code>	Return the position of node in parent children list.
<code>Converter.Internal.getNodeFromPath(t, path)</code>	Return a node from a path.
<code>Converter.Internal.getPathsFromName(node, name)</code>	Return a list of paths corresponding to a given name.
<code>Converter.Internal.getPathsFromType(node, ntype)</code>	Return a list of paths corresponding to a given type.
<code>Converter.Internal.getPathsFromValue(node, value)</code>	Return a list of paths corresponding to a given value.
<code>Converter.Internal.getPathLeaf(path)</code>	Return end of path.

Continued on next page



Table 2.3 – continued from previous page

<code>Converter.Internal.getPathAncestor(path[, level])</code>	Return the path ancestor of path.
<code>Converter.Internal.getZonePaths(t[, pyCGNSLike])</code>	Return the list of paths of zones.
<code>Converter.Internal.getZones(t)</code>	Return a list of all <code>Zone_t</code> nodes.
<code>Converter.Internal.getZonesPerIteration(t[, ...])</code>	Return zones for given iteration of time.
<code>Converter.Internal.getBases(t)</code>	Return a list of all <code>CGNSBase_t</code> nodes.
<code>Converter.Internal.getZoneDim(zone)</code>	Return dimension information from a <code>Zone_t</code> node.
<code>Converter.Internal.getZoneType(zone)</code>	Return 1 for Structured zones, 2 for unstructured zones.

**– Check nodes**

<code>Converter.Internal.printTree(node[, file, ...])</code>	Pretty print a pyTree node.
<code>Converter.Internal.getSizeOf(a)</code>	Return the size of a in octets.
<code>Converter.Internal.checkPyTree(t[, level])</code>	Check pyTree conformity.
<code>Converter.Internal.correctPyTree(t[, level])</code>	Correct a pyTree.

**– Copy nodes**

<code>Converter.Internal.copyRef(node)</code>	Copy a tree sharing node values.
<code>Converter.Internal.copyTree(node[, parent, ...])</code>	Fully copy a tree.
<code>Converter.Internal.copyValue(node[, byName, ...])</code>	Copy the value of nodes specified by <code>byName</code> or <code>byType</code> string.
<code>Converter.Internal.copyNode(node)</code>	Copy only this node (no recursion).

**– Add/remove nodes**

<code>Converter.Internal.append(t, node, path)</code>	Append a node to t specifying its path in t.
<code>Converter.Internal.rmNode(t, node)</code>	Remove given node from t.
<code>Converter.Internal.rmNodeByPath(t, path)</code>	Remove node by specifying its path.
<code>Converter.Internal.rmNodesByName(t, name)</code>	Remove nodes of given name.
<code>Converter.Internal.rmNodesByType(t, ntype)</code>	Remove nodes of given type.

Continued on next page

Table 2.6 – continued from previous page

<code>Converter.Internal.rmNodesByNameAndType(t, ...)</code>	Remove nodes of that match given name and given type at the same time.
<code>Converter.Internal.rmNodesByValue(t, value)</code>	Remove nodes that patch given value.

**– Modify nodes**

<code>Converter.Internal.merge(A[, pathList])</code>	Merge a list of pyTrees in one.
<code>Converter.Internal.renameNode(t, name, newName)</code>	Rename nodes (propagating changes).
<code>Converter.Internal.sortByName(t[, recursive])</code>	Sort nodes by their names (alphabetical order).
<code>Converter.Internal.appendBaseName2ZoneName(t)</code>	Append base name to all zone names.
<code>Converter.Internal.groupBCByBCType(t[, ...])</code>	Tag all BCs of given type with family named FamilyName.

**– Create specific CGNS nodes**

<code>Converter.Internal.newCGNSTree()</code>	Create a new pyTree.
<code>Converter.Internal.newCGNSBase([name, ...])</code>	Create a new Base node.
<code>Converter.Internal.newZone([name, zsize, ...])</code>	Create a new Zone node.
<code>Converter.Internal.newGridCoordinates([...])</code>	Create a GridCoordinates node.
<code>Converter.Internal.newDataArray([name, ...])</code>	Create a new DataArray node.
<code>Converter.Internal.newDataClass([value, parent])</code>	Create a new DataClass node.
<code>Converter.Internal.newDimensionalUnits([...])</code>	Create a new DimensionalUnits node.
<code>Converter.Internal.newDimensionalExponents([...])</code>	Create a new DimensionalExponents node.
<code>Converter.Internal.newDataConversion([...])</code>	Create a new DataConversion node.
<code>Converter.Internal.newDescriptor([name, ...])</code>	Create a new Descriptor node.
<code>Converter.Internal.newGridLocation([value, ...])</code>	Create a new GridLocation node.

Continued on next page

Table 2.8 – continued from previous page

<code>Converter.Internal.newIndexArray([name, ...])</code>	Create a new IndexArray node.
<code>Converter.Internal.newPointList([name, ...])</code>	Create a new PointList node.
<code>Converter.Internal.newPointRange([name, ...])</code>	Create a new PointRange node.
<code>Converter.Internal.newRind([value, parent])</code>	Create a new Rind node.
<code>Converter.Internal.newSimulationType([...])</code>	Create a new SimulationType node.
<code>Converter.Internal.newOrdinal([value, parent])</code>	Create a new Ordinal node.
<code>Converter.Internal.newDiscreteData([name, ...])</code>	Create a new DiscreteData node.
<code>Converter.Internal.newIntegralData([name, ...])</code>	Create a new IntegralData node.
<code>Converter.Internal.newElements([name, ...])</code>	Create a new Elements node.
<code>Converter.Internal.newParentElements([...])</code>	Create a new ParentElements node.
<code>Converter.Internal.newParentElementsPosition([...])</code>	Create a new ParentElementsPosition node.
<code>Converter.Internal.newZoneBC([parent])</code>	Create a new ZoneBC node.
<code>Converter.Internal.newBC([name, pointRange, ...])</code>	Create a new BC node.
<code>Converter.Internal.newBCDataSet([name, ...])</code>	Create BCDataSet node.
<code>Converter.Internal.newBCData([name, parent])</code>	Create a new BCData node.
<code>Converter.Internal.newBCProperty([...])</code>	Create BCProperty node.
<code>Converter.Internal.newAxisymmetry([...])</code>	Create Axisymetry node.
<code>Converter.Internal.newRotatingCoordinates([...])</code>	Create a new RotatingCoordinates node.
<code>Converter.Internal.newFlowSolution([name, ...])</code>	Create a new FlowSolution node.
<code>Converter.Internal.newZoneGridConnectivity([...])</code>	Create a new ZoneGridConnectivity node.
<code>Converter.Internal.newGridConnectivity1to1([...])</code>	Create a new GridConnectivity1to1 node.
<code>Converter.Internal.newGridConnectivity([...])</code>	Create a new GridConnectivity node.

Continued on next page

Table 2.8 – continued from previous page

<code>Converter.Internal. newGridConnectivityType([...])</code>	Create a new <code>GridConnectivityType</code> node.
<code>Converter.Internal. newGridConnectivityProperty([...])</code>	Create a new <code>GridConnectivityType</code> node.
<code>Converter.Internal.newPeriodic([...])</code>	Create a new <code>Periodic</code> node.
<code>Converter.Internal. newOversetHoles([name, ...])</code>	Create a new <code>OversetHoles</code> node.
<code>Converter.Internal. newFlowEquationSet([parent])</code>	Create a new <code>FlowEquationSet</code> node.
<code>Converter.Internal. newGoverningEquations([...])</code>	Create a new <code>GoverningEquation</code> node.
<code>Converter.Internal.newGasModel([value, parent])</code>	Create a new <code>GasModel</code> node.
<code>Converter.Internal. newThermalConductivityModel([...])</code>	Create a new <code>ThermalConductivity</code> node.
<code>Converter.Internal. newViscosityModel([...])</code>	Create a new <code>ViscosityModel</code> node.
<code>Converter.Internal. newTurbulenceClosure([...])</code>	Create a new <code>TurbulenceClosure</code> node.
<code>Converter.Internal. newTurbulenceModel([...])</code>	Create a new <code>TurbulenceModel</code> node.
<code>Converter.Internal. newThermalRelaxationModel([...])</code>	Create a new <code>ThermalRelaxationModel</code> node.
<code>Converter.Internal. newChemicalKineticsModel([...])</code>	Create a new <code>ChemicalKineticsModel</code> node.
<code>Converter.Internal. newEMElectricFieldModel([...])</code>	Create a new <code>EMElectricFieldModel</code> node.
<code>Converter.Internal. newEMConductivityModel([...])</code>	Create a new <code>EMConductivityModel</code> node.
<code>Converter.Internal. newBaseIterativeData([...])</code>	Create a new <code>BaseIterativeData</code> node.
<code>Converter.Internal. newZoneIterativeData([...])</code>	Create a new <code>ZoneIterativeData</code> node.
<code>Converter.Internal. newRigidGridMotion([...])</code>	Create a new <code>RigidMotion</code> node.
<code>Converter.Internal. newRigidGridMotionType([...])</code>	Create a new <code>RigidGridMotionType</code> node.
<code>Converter.Internal. newReferenceState([name, ...])</code>	Create a new <code>Reference State</code> node.
<code>Converter.Internal. newConvergenceHistory([...])</code>	Create a new <code>ConvergenceHistory</code> node.

Continued on next page

Table 2.8 – continued from previous page

---

<code>Converter.Internal.newFamily([name, parent])</code>	Create a new Family node.
<code>Converter.Internal.newFamilyBC([value, parent])</code>	Create a new FamilyBC node.
<code>Converter.Internal.newGeometryReference(...)</code>	Create a new GeometryReference node.
<code>Converter.Internal.newArbitraryGridMotion(...)</code>	Create a new ArbitraryGridMotion node.
<code>Converter.Internal.newUserDefinedData(...)</code>	Create a new UserDefinedData node.
<code>Converter.Internal.newGravity([value, parent])</code>	Create a new Gravity node.

---



## 3.1 Node tests

Converter.Internal.**isTopTree**(node)

Return True if node is a top tree node.

**Parameters** node (a pyTree node) – Input node

**Return type** Boolean: True or False

*Example of use:*

- Tell if a node is a top tree (pyTree):

```
# - isTopTree (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal

t = C.newPyTree(['Base1', 'Base2'])
print(Internal.isTopTree(t))
#>> True
print(Internal.isTopTree(t[2][1]))
#>> False
```

---

Converter.Internal.**isStdNode**(node)

Return 0 if node is a list of standard pyTree nodes, -1 if node is a standard pyTree node, -2 otherwise.

**Parameters** node (a pyTree node) – Input node

**Return type** int: 0: node is a list of standard nodes, -1: node is a standard node, -2: otherwise

*Example of use:*

- Tell if a node is a standard one (pyTree):

```
# - isStdNode (pyTree) -
import Converter.Internal as Internal
import numpy

# This is a standard node
a = ['toto', numpy.zeros(12), [], 'DataArray_t']
print(Internal.isStdNode(a))
#>> -1

# This is not a standard node
b = ['toto', 'tata']
print(Internal.isStdNode(b))
#>> -2

# This is a list of standard nodes
c = ['titi', numpy.zeros(13), [], 'DataArray_t']
print(Internal.isStdNode([a,c]))
#>> 0
```

---

`Converter.Internal.typeOfNode(node)`

Return 1 if node is a zone node, 2 if node is a list of zones, 3 if node is a tree, 4 if node is a base, 5 if node is a list of bases, -1 otherwise.

**Parameters** `node` (a pyTree node) – Input node

**Return type** int: 1: zone node, 2: list of zones, 3: pyTree, 4: base node, 5: list of bases, -1: otherwise

*Example of use:*

- Tell the type of node (pyTree):

```
# - typeOfNode (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal

t = C.newPyTree(['Base1', 'Base2'])
print(Internal.typeOfNode(t))
#>> 3
```

---

`Converter.Internal.isType(node, ntype)`

Compare given type and node type. Wildcards are accepted for ntype.

**Parameters**

- `node` (a pyTree node) – Input node



- **ntype** (string) – CGNS type string ('DataArray\_t', 'Zone\_t', ...)

**Return type** Boolean: True or False

*Example of use:*

- Tell if a node is of given type (pyTree):

```
# - isType (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal

t = C.newPyTree(['Base1', 'Base2'])

# This is true
print(Internal.isType(t, 'CGNSTree_t'))
#>> True

# This is false
print(Internal.isType(t, 'CGNSBase_t'))
#>> False

# Check with wildcard: answer true if the type matches the string
print(Internal.isType(t, 'CGNS*'))
#>> True
```

`Converter.Internal.isName(node, name)`

Compare given name and node name. Wildcards are accepted for name.

#### Parameters

- **node** (a pyTree node) – Input node
- **name** (string) – node name to be checked

**Return type** Boolean: True or False

*Example of use:*

- Tell if a node has given name (pyTree):

```
# - isName (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal

t = C.newPyTree(['Base1', 'Base2'])

# This is false
print(Internal.isName(t[2][1], 'Base3'))
#>> False
```

```
# This is true if node name matches the string
print(Internal.isName(t[2][1], 'Base*'))
#>> True
```

Converter.Internal.**isValue**(node, value)

Compare given value and node value. If value is a string, wildcards are accepted. Accepts also numpy arrays as value.

### Parameters

- **node** (a pyTree node) – Input node
- **value** (int, float, string, numpys) – node value to be checked

**Return type** Boolean: True or False

*Example of use:*

- Tell if a node stores given value (pyTree):

```
# - isValue (pyTree) -
import Converter.Internal as Internal
import numpy

# Check a scalar value
node = Internal.createNode('node1', 'DataArray_t', value=1.)
print(Internal.isValue(node, 1.))
#>> True

# Check a numpy array values
node = Internal.createNode('node1', 'DataArray_t', value=numpy.zeros(10))
print(Internal.isValue(node, numpy.zeros(10)))
#>> True

# Check a string value
node = Internal.createNode('node1', 'DataArray_t', value='toto')
print(Internal.isValue(node, 'toto'))
#>> True
```

Converter.Internal.**isChild**(start, node)

Return true if node is a child of start, even at deep levels. Exists also as isChild1 and isChild2, limited to 1 or 2 recursivity level.

### Parameters

- **start** (a pyTree node) – Input node

- **node** (pyTree node) – node to be checked as a child of start

**Return type** Boolean: True or False

*Example of use:*

- Tell if a node is in the children of another one (pyTree):

```
# - isChild (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal

t = C.newPyTree(['Base1', 'Base2'])

b1 = Internal.getNodeFromPath(t, 'CGNSTree/Base1')
b2 = Internal.getNodeFromPath(t, 'CGNSTree/Base2')
n = Internal.createChild(b1, 'mynode', 'DataArray_t', value=1.)

print(Internal.isChild(b1, n))
#>> True
print(Internal.isChild(b2, n))
#>> False
```

---

**Note:** new in version 2.7.

---

## 3.2 Set/create generic nodes

`Converter.Internal.setName(node, name)`

Set the given name to node (node is modified).

**Parameters**

- **node** (a pyTree node) – Input node
- **name** (string) – node name to be set

**Return type** None

*Example of use:*

- Set given name to node (pyTree):

```
# - setName (pyTree) -
import Converter.Internal as Internal

node = Internal.createNode('node1', 'DataArray_t', value=1.)
```

```
Internal.setName(node, 'myNode'); print(node)
#>> ['myNode', array([ 1.]), [], 'DataArray_t']
```

Converter.Internal.**setType**(node, ntype)

Set the given type to node (node is modified).

### Parameters

- **node** (a pyTree node) – Input node
- **ntype** (string) – node type to be set

**Return type** None

*Example of use:*

- Set given type to node (pyTree):

```
# - setType (pyTree) -
import Converter.Internal as Internal

node = Internal.createNode('node1', 'DataArray_t', value=1.)
Internal.setType(node, 'Zone_t'); print(node)
#>> ['node1', array([ 1.]), [], 'Zone_t']
```

Converter.Internal.**setValue**(node, value=None)

Set the given value in node (node is modified). Value can be provided as an int, a float, a string or a numpy array but it is always stored as a numpy array in node[1].

### Parameters

- **node** (a pyTree node) – Input node
- **value** (int, float, string, numpys) – node value to be set

**Return type** None

*Example of use:*

- Set given value in node (pyTree):

```
# - setValue (pyTree) -
import Converter.Internal as Internal
import numpy

node = Internal.createNode('node1', 'DataArray_t', value=12.)

# Set a scalar value in node
Internal.setValue(node, 1.); print(node)
```

```
#>> ['node1', array([ 1.]), [], 'DataArray_t']

# Set a numpy array in node
Internal.setValue(node, numpy.zeros(10)); print(node)
#>> ['node1', array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]), [],
↪ 'DataArray_t']

# Set an array as a list
Internal.setValue(node, [1.,12.,13.]); print(node)
#>> ['node1', array([ 1., 12., 13.]), [], 'DataArray_t']
```

Converter.Internal.**createNode**(*name*, *ntype*, *value=None*, *children=None*, *parent=None*)

Create a node with a given name and type and optional value and children. If parent is present, created node is also attached to parent node.

#### Parameters

- **name** (string) – created node name
- **ntype** (string) – created node type ('DataArray\_t', 'Zone\_t', ...)
- **value** (int, float, string, numpys) – created node value
- **children** (list of pyTree nodes) – a list of children nodes
- **parent** (pyTree node) – optional node. If given, created node is attached to parent.

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a pyTree node (pyTree):

```
# - createNode (pyTree) -
import Converter.Internal as Internal
import numpy

# Create a node named myNode, of type DataArray_t, with one value 1.
node = Internal.createNode('myNode', 'DataArray_t', value=1., children=[]);
↪ print(node)
#>> ['myNode', array([ 1.]), [], 'DataArray_t']

# Create a node named myNode, of type DataArray_t, with an array valued in a
↪ list
node = Internal.createNode('myNode', 'DataArray_t', value=[12.,14.,15.],
↪ children=[]); print(node)
```

```
#>> ['myNode', array([ 12.,  14.,  15.]), [], 'DataArray_t']

# Create a node named myNode, of type DataArray_t, with a given numpy
a = numpy.zeros( (10) )
a[1] = 1.; a[8] = 2.
node = Internal.createNode('myNode', 'DataArray_t', value=a, children=[]);
↳ print(node)
#>> ['myNode', array([ 0.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  2.,  0.]), [],
↳ 'DataArray_t']

# Create a node named GoverningEquation, of type 'GoverningEquation_t' and
↳ value 'Euler'
node = Internal.createNode('GoverningEquation', 'GoverningEquation_t', value=
↳ 'Euler'); print(node)
#>> ['GoverningEquation', array(['E', 'u', 'l', 'e', 'r'], dtype='|S1'), [],
↳ 'GoverningEquation_t']
```

Converter.Internal.**addChild**(node, child, pos=-1)

Insert a child at given index in the children list of node. If pos=-1, add it at the end.

#### Parameters

- **node** (pyTree node) – modified node
- **child** (pyTree node) – node added as children of node
- **pos** (int) – position of child in children list

**Returns** child (identical to input)

**Return type** pyTree node

*Example of use:*

- Add a child node (pyTree):

```
# - addChild (pyTree) -
import Converter.Internal as Internal

node = Internal.createNode('myNode', 'DataArray_t', value=1.)
child1 = Internal.createNode('child1', 'DataArray_t', value=2.)
child2 = Internal.createNode('child2', 'DataArray_t', value=3.)

# add children nodes to node
Internal.addChild(node, child1, pos=-1) # at the end
Internal.addChild(node, child2, pos=0) # first
print(node)
#>> ['myNode', array([ 1.]), [['child2', array([ 3.]), [], 'DataArray_t'], [
↳ 'child1', array([ 2.]), [], 'DataArray_t']], 'DataArray_t']
```



- **name** (string) – created node name
- **ntype** (string) – created node type
- **value** – created node value
- **children** (list of pyTree nodes) – optional list of nodes to be attached as children to the created node.
- **pos** (int) – position of child in children list

**Returns** created (or modified) node

**Return type** pyTree node

*Example of use:*

- Create a unique child node (pyTree):

```
# - createUniqueChild (pyTree) -
import Converter.Internal as Internal

node = Internal.createNode('myNode', 'DataArray_t', value=1.)
Internal.createUniqueChild(node, 'childName', 'DataArray_t', value=2.)
# Since childName node already exists. Only the value will be set.
Internal.createUniqueChild(node, 'childName', 'DataArray_t', value=3.);
↪ print(node)
#>> ['myNode', array([ 1.]), [['childName', array([ 3.]), [], 'DataArray_t']],
↪ 'DataArray_t']
```

### 3.3 Access nodes

`Converter.Internal.getName(node)`

Return the name of node. Completely equivalent to `node[0]`.

**Parameters** `node` (pyTree node) – input node

**Returns** name of node

**Return type** string

*Example of use:*

- Return name of pyTree node (pyTree):

```
# - getName (pyTree) -
import Converter.Internal as Internal

node = Internal.createNode('myNode', 'DataArray_t', value=1.)
```



```
print(Internal.getName(node))
#>> myNode
```

Converter.Internal.**getType**(node)

Return the type of node. Completely equivalent to node[3].

**Parameters** node (pyTree node) – input node

**Returns** type of node

**Return type** string

*Example of use:*

- Return type of pyTree node (pyTree):

```
# - getType (pyTree) -
import Converter.Internal as Internal

node = Internal.createNode('myNode', 'DataArray_t', value=1.)
print(Internal.getType(node))
#>> DataArray_t
```

Converter.Internal.**getChildren**(node)

Return the type of node. Completely equivalent to node[2].

**Parameters** node (pyTree node) – input node

**Returns** children nodes

**Return type** list of pyTree nodes

*Example of use:*

- Return children of pyTree node (pyTree):

```
# - getChildren (pyTree) -
import Converter.Internal as Internal

node = Internal.createNode('myNode', 'DataArray_t', value=1.)
print(Internal.getChildren(node))
#>> []
```

Converter.Internal.**getValue**(node)

Return value of node. Depending of stored value, return string, int, float or numpy array. It differs from node[1], which is always a numpy array.

**Parameters** `node` (pyTree node) – input node

**Returns** node value

**Return type** string, int, float, numpy array

*Example of use:*

- Return value of pyTree node (`pyTree`):

```
# - getValue of a node (pyTree) -
import Generator.PyTree as G
import Converter.Internal as Internal

# Structured array
a = G.cart((0,0,0), (1., 0.5,1.), (40,50,20))

# Get value stored in a zone node
print(Internal.getValue(a))
#>> [[40 39 0] [50 49 0] [20 19 0]]

# Get type of a zone (from ZoneType node)
node = Internal.getNodeFromName(a, 'ZoneType')

# Print node[1], which is a numpy array
print(node[1])
#>> array(['S', 't', 'r', 'u', 'c', 't', 'u', 'r', 'e', 'd'])

# getValue, return a string in this case
print(Internal.getValue(node))
#>> Structured
```

---

`Converter.Internal.getPath(t, node, pyCGNSLike=False)`

Return path of node in t. Path is a string describing the node names from t to node. For instance: Base/Zone/GridConnectivity. If node is not found, return None.

**Parameters**

- `t` (pyTree node) – starting node
- `node` (pyTree node) – input node
- `pyCGNSLike` (boolean) – if True, paths don't start with CGNSTree

**Returns** path of node in t

**Return type** string

*Example of use:*

- Return the path of node (`pyTree`):

```
# - getPath (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])
print(Internal.getPath(t, a))
#>> CGNSTree/Base/cart
```

`Converter.Internal.getNodesFromName(t, name)`

Return a list of nodes matching given name (wildcards allowed). To accelerate search, this function can be limited to 1, 2 or 3 levels from the starting node using `getNodesFromName1`, `getNodesFromName2`, `getNodesFromName3`. Then wildcards are then no longer allowed.

#### Parameters

- `t` (pyTree node or list of pyTree nodes) – starting node
- `name` (string) – nodes name we are looking for

**Returns** list of nodes that matches name (shared with t)

**Return type** list of pyTree nodes

*Example of use:*

- Return list of nodes specified by a name (pyTree):

```
# - getNodesFromName (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# Return nodes named 'cart'
nodes = Internal.getNodesFromName(t, 'cart'); print(nodes)
#>> [['cart', array([..]), [..], 'Zone_t']]

# Return the 3 coordinate nodes
nodes = Internal.getNodesFromName(t, 'Coordinate*'); print(nodes)
#>> [['CoordinateX', array([..]), [], 'DataArray_t'], ['CoordinateY', array([..]),
↪ 'DataArray_t'], ['CoordinateX', array([..]), [], 'DataArray_t']]
```

Converter.Internal.**getNodeFromName**(t, name)

Return the first node found of given name. Starting node must be a standard pyTree node. Wildcards are NOT accepted. To accelerate search, this function can be limited to 1, 2 or 3 levels from the starting node using getNodeFromName1, getNodeFromName2, getNodeFromName3. If not found, it returns None. This is a fast routine.

### Parameters

- **t** (pyTree node) – starting node
- **name** (string) – node name we are looking for

**Returns** node that matches name (shared with t)

**Return type** pyTree node

*Example of use:*

- Return the first found node specified by a name (pyTree):

```
# - getNodeFromName (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])

# Return the node named 'cart'
node = Internal.getNodeFromName(t, 'cart'); print(node)
#>> ['cart', array([..]), [..], 'Zone_t']
```

---

Converter.Internal.**getByName**(t, name, recursive=-1)

Return a standard node containing the list of matching name nodes as children. Wildcards are accepted.

### Parameters

- **t** (pyTree node or list of pyTree nodes) – starting node
- **name** (string) – node name we are looking for
- **recursive** (int) – if set to 1,2,3, search is limited to 1,2,3 levels. If set to -1, the search is not limited.

**Returns** standard node with children list set as list of nodes that match name

**Return type** pyTree node

*Example of use:*

- Return list of nodes specified by name as a standard node (pyTree):

```
# - getByName (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# Return a standard node containing nodes of name 'cart' as children
node = Internal.getByName(t, 'cart'); print(node)
#>> ['cart', None, [['cart', array(...), [...], 'Zone_t']], None]
```

### Converter.Internal.getNodesFromType(t, ntype)

Return a list of nodes matching given type. To accelerate search, this function can be limited to 1, 2 or 3 levels from the starting node using `getNodesFromType1`, `getNodesFromType2`, `getNodesFromType3`.

#### Parameters

- **t** (pyTree node or list of pyTree nodes) – starting node
- **ntype** (string) – node type we are looking for

**Returns** list of nodes that matches given type (shared with t)

**Return type** list of pyTree nodes

*Example of use:*

- Return list of nodes specified by a type (pyTree):

```
# - getNodesFromType (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])

# Return nodes of type 'Zone_t'
zones = Internal.getNodesFromType(t, 'Zone_t'); print(zones)
#>> [['cart', array(..), [...], 'Zone_t']]

# Limit search to 2 levels (faster)
zones = Internal.getNodesFromType2(t, 'Zone_t'); print(zones)
#>> [['cart', array(..), [...], 'Zone_t']]
```

Converter.Internal.**getNodeFromType**(*t*, *ntype*)

Return the first node found of given type. Starting node must be a standard pyTree node. To accelerate search, this function can be limited to 1, 2 or 3 levels from the starting node using `getNodeFromType1`, `getNodeFromType2`, `getNodeFromType3`. If not found, it returns None. This is a fast routine.

**Parameters**

- **t** (pyTree node) – starting node
- **ntype** (string) – node type we are looking for

**Returns** node that matches type (shared with t)

**Return type** pyTree node

*Example of use:*

- Return node specified by a type (pyTree):

```
# - getNodeFromType (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
t = C.newPyTree(['Base',a])

# Return the first node of type 'Zone_t'
node = Internal.getNodeFromType(t, 'Zone_t'); print(node)

# Limit search to second level (faster)
node = Internal.getNodeFromType2(t, 'Zone_t'); print(node)
```

Converter.Internal.**getByType**(*t*, *ntype*, *recursive=-1*)

Return a standard node containing the list of matching type nodes as children.

**Parameters**

- **t** (pyTree node or list of pyTree nodes) – starting node
- **ntype** (string) – node type we are looking for
- **recursive** (int) – if set to 1,2,3, search is limited to 1,2,3 levels. If set to -1, the search is not limited.

**Returns** standard node with children list set as list of nodes that match type

**Return type** pyTree node*Example of use:*

- Return list of nodes specified by type as a standard node (pyTree):

```
# - getByType (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
t = C.newPyTree(['Base',a])

# Return a standard node containing nodes of type 'Zone_t' as children
zones = Internal.getByType(t, 'Zone_t'); print(zones)
```

Converter.Internal.**getNodesFromNameAndType**(t, name, ntype)

Return a list of nodes matching given name and type. Wildcards are accepted for name and type.

**Parameters**

- **t** (pyTree node or list of pyTree nodes) – starting node
- **name** (string) – node name we are looking for
- **ntype** (string) – node type we are looking for

**Returns** list of nodes that matches given name and type (shared with t)

**Return type** list of pyTree nodes

*Example of use:*

- Return list of nodes specified by a name and type (pyTree):

```
# - getNodesFromNameAndType (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# Return nodes named 'cart' and of type 'Zone_t'
nodes = Internal.getNodesFromNameAndType(t, 'cart', 'Zone_t'); print(nodes)
#>> [['cart', array([..]), [..], 'Zone_t']]

# Return the 3 coordinate nodes
```

```
nodes = Internal.getNodesFromNameAndType(t, 'Coordinate*', 'Data*_t');  
↳ print(nodes)  
#>> [['CoordinateX', array([..]), [], 'DataArray_t'], ['CoordinateY', array([]),  
↳ 'DataArray_t'], ['CoordinateX', array([..]), [], 'DataArray_t']]
```

---

**Note:** new in version 2.7.

---

Converter.Internal.**getNodeFromNameAndType**(t, name, ntype)

Return the first node found of given name and type. Starting node must be a standard pyTree node. If not found, it returns None.

### Parameters

- **t** (pyTree node) – starting node
- **name** (string) – node name we are looking for
- **ntype** (string) – node type we are looking for

**Returns** node that matches name and type (shared with t)

**Return type** pyTree node

*Example of use:*

- Return node specified by a name and type (pyTree):

```
# - getNodeFromNameAndType (pyTree) -  
import Converter.PyTree as C  
import Generator.PyTree as G  
import Converter.Internal as Internal  
  
a = G.cart((0,0,0), (1,1,1), (10,10,10))  
t = C.newPyTree(['Base', a])  
  
# Return the node named 'cart' and type 'Zone_t'  
node = Internal.getNodeFromNameAndType(t, 'cart', 'Zone_t'); print(node)  
#>> ['cart', array([..]), [..], 'Zone_t']
```

---

**Note:** new in version 2.7.

---

Converter.Internal.**getNodesFromValue**(t, value)

Return a list of nodes matching given value. if value is a string, wildcards are accepted.



**Parameters**

- **t** (pyTree node or list of pyTree nodes) – starting node
- **value** (string, int, float, numpy) – node value we are looking for

**Returns** list of nodes that match given value (shared with t)

**Return type** list of pyTree nodes

*Example of use:*

- Return list of nodes specified by value (pyTree):

```
# - getNodesFromValue (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
t = C.newPyTree(['Base',a])

# Return nodes with given value
nodes = Internal.getNodesFromValue(t, 2.4); print(nodes)
#>> []
nodes = Internal.getNodesFromValue(t, 'Structured'); print(nodes)
#>> [['ZoneType', array(..), [], 'ZoneType_t']]
```

Converter.Internal.**getParentOfNode**(t, node)

Return the parent node of given node in t. t must be a higher node in the tree. It returns p (the parent node) and c (the position number of node in the parent's children list). If t is a node of a pyTree, then p[2][c] = node. If t is a list of standard nodes, then p == t and p[c] = node. If node is not found, then p is None. This is an expansive routine. Prefer top-down tree traversal, if possible.

**Parameters**

- **t** (pyTree node) – higher node
- **node** (pyTree node) – node the parent of which is looked for

**Returns** parent node and position in children list

**Return type** tuple (pyTree node, c)

*Example of use:*

- Return parent node (pyTree):

```
# - getParentOfNode (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])

(p, c) = Internal.getParentOfNode(t, a); print(p)
#>> ['Base', array(..), [..], 'CGNSBase_t']
```

---

Converter.Internal.**getParentFromType**(*t, node, parentType*)

Return the parent node of given node in t. The parent can be any level upper. t must be a higher node in the tree. It returns the first parent node matching that type.

**Parameters**

- **t** (pyTree node) – higher node
- **node** (pyTree node) – node the parent of which is looked for
- **parentType** (string) – type of parent

**Returns** parent node

**Return type** pyTree node

*Example of use:*

- Return parent node of given type (pyTree):

```
# - getParentFromType (pyTree) -
import Converter.Internal as Internal

a = Internal.createNode('level0', 'DataArray_t', 0)
b = Internal.createChild(a, 'level1', 'DataArray_t', 1)
c = Internal.createChild(b, 'level2', 'DataArray_t', 2)

p = Internal.getParentFromType(a, c, 'DataArray_t'); print(p[0])
#>> level1
```

---

**Note:** new in version 2.7.

---

Converter.Internal.**getParentsFromType**(*t, node, parentType*)

Return a list of parent nodes of given node in *t* of given type. The parent can be any level upper. *t* must be a higher node in the tree.

**Parameters**

- **t** (pyTree node) – higher node
- **node** (pyTree node) – node the parent of which is looked for
- **parentType** (string) – type of parent

**Returns** parent node list

**Return type** list of pyTree nodes

*Example of use:*

- Return parent nodes of given type (pyTree):

```
# - getParentsFromType (pyTree) -
import Converter.Internal as Internal

a = Internal.createNode('level0', 'DataArray_t', 0)
b = Internal.createChild(a, 'level1', 'DataArray_t', 1)
c = Internal.createChild(b, 'level2', 'DataArray_t', 2)

p = Internal.getParentsFromType(a, c, 'DataArray_t')
for i in p: print(i[0])
#>> level0
#>> level1
```

---

**Note:** new in version 2.7.

---

Converter.Internal.**getNodePosition**(*node, parent*)

Return the position of node in parent children list. If node is not found, return -1.

**Parameters**

- **node** (pyTree node) – node which position need to be found
- **parent** (pyTree node) – the parent of node

**Returns** node position in parent children

**Return type** int

*Example of use:*

- Return position of a node (pyTree):

```
# - getNodePosition (pyTree) -
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
g = Internal.getNodeFromType(a, 'GridCoordinates_t')
x = Internal.getNodeFromName(g, 'CoordinateZ')

print(Internal.getNodePosition(x, g))
#>> 2
```

---

**Note:** new in version 2.9.

---

Converter.Internal.**getNodeFromPath**(*t*, *path*)

Return a node from its path string. Note that node path is relative to input node. If not found, it returns None.

**Parameters**

- **t** (pyTree node) – starting node
- **path** (string) – path ('Base/Zone')

**Returns** found node (shared with t)

**Return type** pyTree node

*Example of use:*

- Return node from path (pyTree):

```
# - getNodeFromPath (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])

# Return GridCoordinates node
coords = Internal.getNodeFromPath(t, 'Base/cart/GridCoordinates'); print(coords)
#>> ['GridCoordinates', None, [...], 'DataArray_t']

# Return GridCoordinates node (path is relative to input node)
coords = Internal.getNodeFromPath(a, 'GridCoordinates'); print(coords)
#>> ['GridCoordinates', None, [...], 'DataArray_t']
```

---

Converter.Internal.**getPathsFromName**(*node*, *name*, *pyCGNSLike=False*)

Return a list of paths corresponding to a given name. The paths are built from node. Wildcards are accepted for name.

**Parameters**

- **node** (pyTree node or list of pyTree nodes) – starting node
- **name** – node name we are looking for
- **pyCGNSLike** (boolean) – if True, paths don't start with CGNSTree

**Returns** list of paths of nodes matching given name

**Return type** list of strings

*Example of use:*

- Return list of paths specified by name (pyTree):

```
# - getPathsFromName (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# Return paths of zone named 'cart'
paths = Internal.getPathsFromName(t, 'cart'); print(paths)
#>> ['CGNSTree/Base/cart']

# Return the 3 coordinate paths
paths = Internal.getPathsFromName(t, 'Coordinate*'); print(paths)
#>> ['CGNSTree/Base/cart/GridCoordinates/CoordinateX', '/CGNSTree/Base/cart/
↪GridCoordinates/CoordinateY', '/CGNSTree/Base/cart/GridCoordinates/CoordinateZ
↪']
```

---

**Note:** new in version 2.5.

---

Converter.Internal.**getPathsFromType**(*node*, *ntype*, *pyCGNSLike=False*)

Return a list of paths corresponding to a given type. The paths are built from node.

**Parameters**

- **node** (pyTree node or list of pyTree nodes) – starting node

- **ntype** (string) – node type we are looking for
- **pyCGNSLike** (boolean) – if True, paths don't start with CGNSTree

**Returns** list of paths of nodes matching given type

**Return type** list of strings

*Example of use:*

- Return list of paths specified by type (pyTree):

```
# - getPathsFromType (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# Return nodes of type 'Zone_t'
zonePaths = Internal.getPathsFromType(t, 'Zone_t'); print(zonePaths)
#>> ['CGNSTree/Base/cart']
```

---

`Converter.Internal.getPathsFromValue(node, value, pyCGNSLike=False)`

Return a list of paths corresponding to a given value. The paths are built from node. If value is a string, wildcards are accepted.

**Parameters**

- **node** (pyTree node or list of pyTree nodes) – starting node
- **value** (string, int, float, numpy) – node value we are looking for
- **pyCGNSLike** (boolean) – if True, paths don't start with CGNSTree

**Returns** list of paths of nodes matching given type

**Return type** list of strings

*Example of use:*

- Return list of paths specified by value (pyTree):

```
# - getPathsFromValue (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
```

```
t = C.newPyTree(['Base', a])

# Return nodes with given value
paths = Internal.getPathsFromValue(t, 3.1); print(paths)
#>> ['CGNSTree/CGNSLibraryVersion']
paths = Internal.getPathsFromValue(t, 'Structured'); print(paths)
#>> ['CGNSTree/Base/cart/ZoneType']
```

---

**Note:** New in version 2.5

---

Converter.Internal.**getPathLeaf**(*path*)

Return the last term of path.

**Parameters** *path* (string) – input path

**Returns** last term of path

**Return type** string

*Example of use:*

- Return leaf of path (pyTree):

```
# - getPathLeaf (pyTree) -
import Converter.Internal as Internal

print(Internal.getPathLeaf('CGNSTree/Base/Zone'))
#>> Zone
```

---

**Note:** New in version 2.5

---

Converter.Internal.**getPathAncestor**(*path*, *level=-1*)

Return the path ancestor of path.

**Parameters**

- *path* (string) – input path
- *level* (int) – number of level to go up

**Returns** path of ancestor

**Return type** string

*Example of use:*

- Return ancestor path (pyTree):

```
# - getPathAncestor (pyTree) -
import Converter.Internal as Internal

print(Internal.getPathAncestor('CGNSTree/Base/Zone', level=1))
#>> CGNSTree/Base
```

---

**Note:** New in version 2.5

---

Converter.Internal.**getZonePaths**(t, pyCGNSLike=False)

Return the list of paths of Zone\_t nodes. Paths are relative to node t.

**Parameters**

- t (pyTree node or list of pyTree nodes) – starting node
- pyCGNSLike (boolean) – if True, paths don't start with CGNSTree

**Returns** list of Zone\_t paths

**Return type** list of strings

*Example of use:*

- Return zone paths of tree (pyTree):

```
# - getZonePaths (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])

# Return paths of 'Zone_t'
zonePaths = Internal.getZonePaths(t); print(zonePaths)
#>> ['CGNSTree/Base/cart']
```

---

**Note:** New in version 2.5

---

Converter.Internal.**getZones**(t)

Return the list of Zone\_t nodes.

**Parameters** t (pyTree node or list of pyTree nodes) – starting node



**Returns** list of Zone\_t nodes (shared with t)

**Return type** list of pyTree nodes

*Example of use:*

- Return zone nodes of tree (pyTree):

```
# - getZones (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])

# Return nodes of type 'Zone_t'
zones = Internal.getZones(t); print(zones)
#>> [['cart', array(..), [..], 'Zone_t']]
```

Converter.Internal.**getZonesPerIteration**(t, iteration=None, time=None)

Return the list of Zone\_t nodes matching a given iteration. A BaseIterativeData node must exist in input pyTree. If iteration is provided, return a list of zones matching iteration. If time is provided, return a list of zones matching time. If none is set, return a list of list of zones for each iterations.

**Parameters**

- **t** (pyTree node or list of pyTree nodes) – starting node
- **iteration** (int) – iteration number
- **time** (float) – desired time

**Returns** list of Zone\_t nodes (shared with t) or list of list of zones

**Return type** list of pyTree nodes

*Example of use:*

- Return zone nodes of tree of given iteration (pyTree):

```
# - getZonesPerIterations (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])
b = Internal.getNodeFromName(t, 'Base')
```

```
n = Internal.newBaseIterativeData(name='BaseIterativeData', nsteps=1, itype=
↳ 'IterationValues', parent=b)
Internal.newDataArray('TimeValues', value=[0.], parent=n)
Internal.newDataArray('NumberOfZones', value=[1], parent=n)
Internal.newDataArray('ZonePointers', value=[[ 'cart' ]], parent=n)

# List of zones of it=0
zones = Internal.getZonesPerIteration(t, iteration=0)
# List of zones of time 0.
zones = Internal.getZonesPerIteration(t, time=0.)
# All zones of all iterations
zones = Internal.getZonesPerIteration(t)
```

Converter.Internal.**getBases**(t)

Return the list of CGNSBase\_t nodes.

**Parameters** t (pyTree node or list of pyTree nodes) – starting node

**Returns** list of CGNSBase\_t nodes (shared with t)

**Return type** list of pyTree nodes

*Example of use:*

- Return base nodes of tree (pyTree):

```
# - getBases (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base', a])

# Return nodes of type 'Zone_t'
bases = Internal.getBases(t); print(bases)
#>> [['Base', array(..), [..], 'CGNSBase_t']]
```

Converter.Internal.**getZoneDim**(zone)

Return the dimension of a zone node. Return ['Structured', ni, nj, nk, celldim] for structured grids, return ['Unstructured', np, ne, eltsName, celldim] for unstructured grids. np is the number of points, ne the number of elements, celldim is 0, 1, 2, 3 depending on element dimension, eltsName is one of NODE, BAR, TRI, QUAD, TETRA, PYRA, PENTA, NGON, MULTIPLE.

**Parameters** zone (pyTree node of type 'Zone\_t') – a 'Zone\_t' node

**Returns** ['Structured', ni, nj, nk, celldim] or ['Unstructured', np, ne, elt-  
sName, celldim]

**Return type** list of data

*Example of use:*

- Return zone dimension information (pyTree):

```
# - getZoneDim (pyTree) -
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
dim = Internal.getZoneDim(a); print(dim)
#>> ['Structured', 10, 10, 10, 3]

a = G.cartTetra((0,0,0), (1,1,1), (10,10,10))
dim = Internal.getZoneDim(a); print(dim)
#>> ['Unstructured', 1000, 3645, 'TETRA', 3]
```

`Converter.Internal.getZoneType(zone)`

Return the nature of zone. Return 1 if zone is structured, 2 if unstructured, 0 if failed.

**Parameters** `zone` (pyTree node of type 'Zone\_t') – a 'Zone\_t' node

**Returns** 1 or 2

**Return type** int

*Example of use:*

- Return zone nature (pyTree):

```
# - getZoneType (pyTree) -
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
print(Internal.getZoneType(a))
#>> 1

a = G.cartTetra( (0,0,0), (1,1,1), (10,10,10) )
print(Internal.getZoneType(a))
#>> 2
```

## 3.4 Check nodes

`Converter.Internal.printTree(node, file=None, stdout=None, editor=None, color=False)`

Pretty print a pyTree or a pyTree node to screen or in file.

### Parameters

- **node** (pyTree node or list of pyTree nodes) – input node
- **file** (string) – file name (optional)
- **stdout** (file object) – a file object (as created by open)(optional)
- **editor** (string) – an editor name (optional)
- **color** (True or False) – trigger colored output with ANSI codes (optional)

*Example of use:*

- Pretty print pyTree nodes (pyTree):

```
# - printTree (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal
import Generator.PyTree as G

a = G.cart((0,0,0), (1,1,1), (10,10,10))
Internal.printTree(a) # can print a zone (or any node or any list of nodes)
#>> ['cart',array(shape=(3, 3),dtype='int32',order='F'),[2 sons],'Zone_t']
#>>  |['_ZoneType',array('Structured',dtype='|S1'),[0 son],'ZoneType_t']
#>>  |['_GridCoordinates',None,[3 sons],'GridCoordinates_t']
#>>      |['_CoordinateX',array(shape=(10, 10, 10),dtype='float64',order='F'),
↪[0 son],'DataArray_t']
#>>      |['_CoordinateY',array(shape=(10, 10, 10),dtype='float64',order='F'),
↪[0 son],'DataArray_t']
#>>      |['_CoordinateZ',array(shape=(10, 10, 10),dtype='float64',order='F'),
↪[0 son],'DataArray_t']

t = C.newPyTree(['Base',a])
Internal.printTree(t) # can print a tree
#>> ['CGNSTree',None,[2 sons],'CGNSTree_t']
#>>  |['_CGNSLibraryVersion',array([3.1],dtype='float64'),[0 son],
↪'_CGNSLibraryVersion_t']
#>>  ..

Internal.printTree(t, file='toto.txt') # in a file
f = open('toto.txt', 'a')
```

```
Internal.printTree(t, stdout=f) # in a file object
Internal.printTree(t, editor='emacs') # with an editor
```

Converter.Internal.**getSizeOf**(node)

Return the size of input node and attached nodes in octets.

**Parameters** node (pyTree node of list of pyTree nodes) – input node

**Returns** size of node in octets

**Return type** int

*Example of use:*

- Get size of node (pyTree):

```
# - getSizeOf (pyTree) -
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
print(Internal.getSizeOf(a))
```

Converter.Internal.**checkPyTree**(t, level=-20)

Check pyTree t following level (0: valid version node, 1: node conformity, 2: unique base name, 3: unique zone name, 4: unique BC name, 5: valid BC range, 6: valid opposite BC range for match and nearmatch, 7: referenced familyZone and familyBCs must be defined in bases, 8: valid CGNS types, 9: valid connectivity, 10: valid CGNS flowfield name and dimension). If level=-20, all previous checks are performed.

Return a list of pairs of invalid nodes and error message.

**Parameters**

- **t** (pyTree node or list of pyTree nodes) – input pyTree
- **level** (int) – level of check

**Returns** [(node, 'error message')]

**Return type** list of tuple (node, 'error')

*Example of use:*

- Check pyTree (pyTree):

```
# - checkPyTree (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
b = G.cart((9,0,0), (1,1,1), (10,10,10))
c = G.cartTetra((9,9,0), (1,1,1), (10,10,10))
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'imin')
t = C.newPyTree(['Base',a,b,c])
t = X.connectMatch(t)

errors = []
# check unique base names
errors += Internal.checkPyTree(t, level=2)
# check unique zone names
errors += Internal.checkPyTree(t, level=3)
# check unique BC names
errors += Internal.checkPyTree(t, level=4)
# check BC ranges
errors += Internal.checkPyTree(t, level=5)
# check opposite ranges
errors += Internal.checkPyTree(t, level=6)
# check family definition
errors += Internal.checkPyTree(t, level=7)
# check CGNSTypes
errors += Internal.checkPyTree(t, level=8)
# check element nodes
errors += Internal.checkPyTree(t, level=9); print(errors)
#>> []

# Introduce errors (on purpose!)
n = Internal.getNodeFromType(t, 'Zone_t')
Internal.setType(n, 'Zon_t')
errors = Internal.checkPyTree(t, level=8); print(errors)
#>> [['cart', array(..), [..], 'Zon_t'], 'Unknown CGNS type Zon_t for node cart.
↪\n']
```

---

Converter.Internal.**correctPyTree**(t, level=-20)

Correct a pyTree t following level (0: valid version node, 1: node conformity, 2: unique base name, 3: unique zone name, 4: unique BC name, 5: valid BC range, 6: valid opposite BC range for match and nearmatch, 7: referenced familyZone and familyBCs must be defined in bases, 8: valid CGNS types, 9: valid connectivity, 10: valid CGNS flowfield name and dimension).

Generally invalid nodes are suppressed. If level=-20, all previous checks are performed.

Exists also as in place version (`_correctPyTree`) that modifies `t` and returns `None`.

**Parameters**

- `t` (pyTree node or list of pyTree nodes) – input pyTree
- `level` (int) – level of check

**Returns** modified reference copy of `t`

**Return type** same as input

*Example of use:*

- Correct pyTree (pyTree):

```
# - correctPyTree (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Connector.PyTree as X
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
b = G.cart((9,0,0), (1,1,1), (10,10,10))
a = C.addBC2Zone(a, 'wall1', 'BCWall', 'imin')
t = C.newPyTree(['Base', a, b])
t = X.connectMatch(t)

t = Internal.correctPyTree(t)
```

## 3.5 Copy nodes

`Converter.Internal.copyRef(node)`

Copy recursively sharing node values (in particular data numpys are shared).

**Parameters** `node` (pyTree node or list of pyTree nodes) – input data

**Returns** copy of input data

**Return type** same as input

*Example of use:*

- Copy pyTree with references (pyTree):

```
# - copyRef (pyTree) -
import Converter.PyTree as C
```

```
import Converter.Internal as Internal
import Generator.PyTree as G

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# t2 and t are sharing values (also data numpys)
t2 = Internal.copyRef(t)
```

---

Converter.Internal.**copyTree**(node)

Fully copy a tree. Node values (in particular data numpys) are copied.

**Parameters** node (pyTree node or list of pyTree nodes) – input data

**Returns** copy of input data

**Return type** same as input

*Example of use:*

- Copy pyTree (pyTree):

```
# - copyTree (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# t2 is a full copy of t (values/numpys are also copied)
t2 = Internal.copyTree(t)
```

---

Converter.Internal.**copyValue**(node, byName=None, byType=None)

Copy recursively the value of nodes specified by byName or byType string. Wildcards are accepted.

**Parameters**

- **node** (pyTree node or list of pyTree nodes) – input data
- **byName** (string) – name of nodes to be copied or None
- **byType** (string) – type of nodes to be copied or None

**Returns** copy of input data

**Return type** same as input



*Example of use:*

- Copy value (pyTree):

```
# - copyValue (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal
import Generator.PyTree as G

a = G.cart((0,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a])

# t2 has numpy copy only for nodes of type 'DataArray_t'
t2 = Internal.copyValue(t, byType='DataArray_t')

# t3 has numpy copy only for nodes of name 'Coordinate*'
t3 = Internal.copyValue(t, byName='Coordinate*')
```

`Converter.Internal.copyNode(node)`

Copy only this node (no recursion). Node value (in particular data numpys) is copied.

**Parameters** `node` (pyTree node or list of pyTree nodes) – input node

**Returns** copy of input node

**Return type** same as input

*Example of use:*

- Copy node (pyTree):

```
# - copyNode (pyTree) -
import Converter.Internal as Internal

a = Internal.newDataArray(value=[1,2,3])
# Copy only the numpy of this node
b = Internal.copyNode(a)
# Modify numpy of b
b[1][0]=5
# a is not modified
print(a)
#>> ['Data', array([1, 2, 3], dtype=int32), [], 'DataArray_t']
print(b)
#>> ['Data', array([5, 2, 3], dtype=int32), [], 'DataArray_t']
```

## 3.6 Add/remove node

`Converter.Internal.append(t, node, path)`

Append a node by specifying its path. Note that the path is relative to t.

Exists also as in place version (`_append`) that modifies t and returns None.

### Parameters

- **t** (pyTree node) – starting node
- **node** (pyTree node) – node to be added
- **path** – the path where node should be added ('Base/')

**Returns** modified reference copy of t

**Return type** same as t

*Example of use:*

- Append a node by its path (pyTree):

```
# - append (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

t = C.newPyTree(['Base', 'Base2'])
a = G.cart((0,0,0), (1,1,1), (10,10,10))

# Append a to 'Base' node of t
t = Internal.append(t, a, 'Base')

#>> ['CGNSTree',None,[3 sons],'CGNSTree_t']
#>> |['_CGNSLibraryVersion',array([3.1],dtype='float64'),[0 son],
↳'_CGNSLibraryVersion_t']
#>> |['_Base',array(shape=(2,),dtype='int32',order='F'),[1 son],'CGNSBase_t']
#>> | |['_cart',array(shape=(3, 3),dtype='int32',order='F'),[2 sons],'Zone_
↳t']
#>> | ...
#>> |['_Base2',array(shape=(2,),dtype='int32',order='F'),[0 son],'CGNSBase_t
↳']
C.convertPyTree2File(t, 'out.cgns')
```

`Converter.Internal.rmNode(t, node)`

Remove given node in t. t is modified.

### Parameters

- **t** (pyTree node) – starting node
- **node** (pyTree node) – node to be removed

**Return type** None

*Example of use:*

- Remove a node (pyTree):

```
# - _rmNode (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

t = C.newPyTree(['Base', 'Base2'])
a = G.cart((0,0,0), (1,1,1), (10,10,10))
t[2][1][2] += [a]

# rm the node a from t
Internal._rmNode(t, a)

#>> ['CGNSTree',None,[3 sons],'CGNSTree_t']
#>>  |['_CGNSLibraryVersion',array([3.1],dtype='float64'),[0 son],
↪'_CGNSLibraryVersion_t']
#>>  |['_Base',array(shape=(2,),dtype='int32',order='F'),[0 son],'CGNSBase_t']
#>>  |['_Base2',array(shape=(2,),dtype='int32',order='F'),[0 son],'CGNSBase_t']
```

`Converter.Internal.rmNodeByPath(t, path)`

Remove node that corresponds to path in t.

Exists also as in place version (`_rmNodeByPath`) that modifies t and returns None.

**Parameters**

- **t** (pyTree node) – input node
- **path** (string) – path of node to be removed (ex: 'Base/Zone0')

**Returns** reference copy of t with node removed

**Return type** pyTree node

*Example of use:*

- Remove node by its path (pyTree):

```
# - rmNodeByPath (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal
```

```
t = C.newPyTree(['Base', 'Base2'])
for i in range(10):
    a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
    a[0] = 'Cart'+str(i)
    t[2][1][2].append(a)

t = Internal.rmNodeByPath(t, 'Base/Cart2')
t = Internal.rmNodeByPath(t, 'Base/Cart4')
C.convertPyTree2File(t, 'out.cgns')
```

---

Converter.Internal.**rmNodesByName**(t, name)

Remove all nodes in t that match given name.

Exists also as in place version (`_rmNodesByName`) that modifies t and returns None.  
Exists also as in place with search limited to 1 or 2 levels as `_rmNodesByName1` and `_rmNodesByName2`.

#### Parameters

- **t** (pyTree node or list of pyTree nodes) – input node
- **name** (string) – name of nodes to be removed (wildcards are accepted)

**Returns** reference copy of t with nodes removed

**Return type** same as t

*Example of use:*

- Remove nodes that match given name (pyTree):

```
# - rmNodesByName (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

t = C.newPyTree(['Base', 'Base2'])
for i in range(10):
    a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
    a[0] = 'Cart'+str(i)
    t[2][1][2].append(a)

t = Internal.rmNodesByName(t, 'Cart0')
t = Internal.rmNodesByName(t, 'Cart*')
C.convertPyTree2File(t, 'out.cgns')
```

Converter.Internal.**rmNodesByType**(*t*, *ntype*)

Remove all nodes in *t* that match given type.

Exists also as in place version (`_rmNodesByType`) that modifies *t* and returns None.  
Exists also as in place with search limited to 1 or 2 levels as `_rmNodesByType1` and `_rmNodesByType2`.

#### Parameters

- **t** (pyTree node or list of pyTree nodes) – input node
- **ntype** (string) – type of nodes to be removed

**Returns** reference copy of *t* with nodes removed

**Return type** same as *t*

*Example of use:*

- Remove nodes that match given type (pyTree):

```
# - rmNodesByType (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

t = C.newPyTree(['Base', 'Base2'])
for i in range(10):
    a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
    a[0] = 'Cart'+str(i)
    t[2][1][2].append(a)

t = Internal.rmNodesByType(t, 'Zone_t')
C.convertPyTree2File(t, 'out.cgns')
```

Converter.Internal.**rmNodesByNameAndType**(*t*, *name*, *ntype*)

Remove all nodes that match given type and given name at the same time.

Exists also as in place version (`_rmNodesByNameAndType`) that modifies *t* and returns None.

#### Parameters

- **t** (pyTree node or list of pyTree nodes) – input node
- **name** (string) – name of nodes to be removed (wildcards accepted)
- **ntype** (string) – type of nodes to be removed

**Returns** reference copy of *t* with nodes removed

**Return type** same as *t*

*Example of use:*

- Remove nodes that match given name and type (pyTree):

```
# - rmNodesByNameAndType (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

t = C.newPyTree(['Base', 'Base2'])
for i in range(10):
    a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
    a[0] = 'Cart'+str(i)
    t[2][1][2].append(a)

t = Internal.rmNodesByNameAndType(t, 'Cart0', 'Zone_t')
C.convertPyTree2File(t, 'out.cgns')
```

---

`Converter.Internal.rmNodesByValue(t, value)`

Remove all nodes in t that match given value.

Exists also as in place version (`_rmNodesByValue`) that modifies t and returns None.

#### Parameters

- **t** (pyTree node or list of pyTree nodes) – input node
- **value** (string, number...) – value of nodes to be removed

**Returns** reference copy of t with nodes removed

**Return type** same as t

*Example of use:*

- Remove nodes that match given value (pyTree):

```
# - rmNodesByValue (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart( (0,0,0), (1,1,1), (10,10,10) )
a = C.fillEmptyBCWith(a, 'far', 'BCFarfield')

a = Internal.rmNodesByValue(a, 'BCFarfield')
t = C.newPyTree(['Base', a])
C.convertPyTree2File(t, 'out.cgns')
```

## 3.7 Modify nodes

Converter.Internal.**merge**(A)

Merge a list of trees defined in [t1, t2,...]. Return a merged tree. If a node appears more than once in different trees of the list, the first found node is kept.

**Parameters** A (list of pyTrees) – list of trees to be merged

**Returns** merged tree (reference copy)

**Return type** pyTree

*Example of use:*

- Merge trees (pyTree):

```
# - merge (pyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal
import Generator.PyTree as G

t1 = C.newPyTree(['Base1', 2, 'Base2', 3])
t2 = C.newPyTree(['Other1', 'Other2', 'Base2'])
a = G.cart((0,0,0), (1,1,1), (10,10,10)); t1[2][1][2] += [a]
t = Internal.merge([t1, t2])

#>> ['CGNSTree',None,[5 sons],'CGNSTree_t']
#>> |['_CGNSLibraryVersion',array([3.1],dtype='float64'),[0 son],
↳'_CGNSLibraryVersion_t']
#>> |['_Base1',array(shape=(2,),dtype='int32',order='F'),[1 son],'CGNSBase_t']
#>> | |['_cart',array(shape=(3, 3),dtype='int32',order='F'),[2 sons],'Zone_t
↳']
#>> | ...
#>> |['_Base2',array(shape=(2,),dtype='int32',order='F'),[0 son],'CGNSBase_t']
#>> |['_Other1',array(shape=(2,),dtype='int32',order='F'),[0 son],'CGNSBase_t
↳']
#>> |['_Other2',array(shape=(2,),dtype='int32',order='F'),[0 son],'CGNSBase_t
↳']
C.convertPyTree2File(t, 'out.cgns')
```

Converter.Internal.**renameNode**(t, name, newName)

Rename node named 'name' with 'newName' in t. Occurances of name elsewhere in the tree t is replaced with 'newName'. Wildcards are accepted for name.

Exists also as in place version (`_renameNode`) that modifies t and returns None.

**Parameters** t (pyTree node or list of pyTree nodes) – input node

**Returns** reference copy of t

**Return type** same as t

*Example of use:*

- Rename node (pyTree):

```
# - renameNode (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal
import Connector.PyTree as X

a = G.cart((0,0,0), (1,1,1), (10,10,10))
b = G.cart((9,0,0), (1,1,1), (10,10,10))
t = C.newPyTree(['Base',a,b])
t = X.connectMatch(t)

# Change the zone named 'cart' and its reference in BMatch (GridConnectivity)
t = Internal.renameNode(t, 'cart', 'myCart')
C.convertPyTree2File(t, 'out.cgns')
```

---

Converter.Internal.**sortByName**(t, recursive=True)

Sort nodes by their name (alphabetical order). If recursive=True, sort also children nodes.

Exists also as in place version (`_sortByName`) that modifies t and returns None.

#### Parameters

- **t** (pyTree node or list of pyTree nodes) – starting pyTree node
- **recursive** (boolean) – if True, perform sort also on all children nodes

**Returns** reference copy of t

**Return type** same as t

*Example of use:*

- Sort nodes by name (pyTree):

```
# - sortByName (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10)); a[0] = 'b'
b = G.cart((12,0,0), (1,1,1), (10,10,10)); b[0] = 'a'
t = C.newPyTree(['Base',a,b])
```



```
t = Internal.sortByName(t)
C.convertPyTree2File(t, 'out.cgns')
```

Converter.Internal.**appendBaseName2ZoneName**(*t*, *updateRef=True*, *separator='\_'*,  
*trailing=""*)

Append base name to zone name (resulting name is `baseName_zoneName` for each zone). If `updateRef=True`, reference to zone names in BC,... are replaced. Separator between base and zone name can be set with `separator`, a trailing string can also be added.

Exists also as in place version (`_appendBaseName2ZoneName`) that modifies `t` and returns `None`.

#### Parameters

- **t** (top pyTree node or list of Base nodes) – input data
- **updateRef** (Boolean) – True if reference to zone name has to be changed
- **separator** (string) – separator between base and zone name
- **trailing** (string) – trailing to be optionally added to merge name

**Returns** reference copy of input

**Return type** same as input

*Example of use:*

- Append base name to zone name (pyTree):

```
# - appendBaseName2ZoneName (pyTree) -
import Converter.PyTree as C
import Generator.PyTree as G
import Converter.Internal as Internal

a = G.cart((0,0,0), (1,1,1), (10,10,10)); a[0] = 'a'
b = G.cart((11,0,0), (1,1,1), (10,10,10)); b[0] = 'b'
t = C.newPyTree(['Base',a,b])

t = Internal.appendBaseName2ZoneName(t)

#>> ['CGNSTree',None,[2 sons],'CGNSTree_t']
#>> |['_CGNSLibraryVersion',array([3.1],dtype='float64'),[0 son],
↪ 'CGNSLibraryVersion_t']
#>> |['_Base',array(shape=(2,),dtype='int32',order='F'),[2 sons],'CGNSBase_t
↪ ']
#>> |['_Base_a',array(shape=(3, 3),dtype='int32',order='F'),[2 sons],
↪ 'Zone_t']
```

```

#>>      |   |_['ZoneType',array('Structured',dtype='|S1'),[0 son], 'ZoneType_t
↪']
#>>      |   ...
#>>      |_['Base_b',array(shape=(3, 3),dtype='int32',order='F'),[2 sons],
↪'Zone_t']
#>>      |_['ZoneType',array('Structured',dtype='|S1'),[0 son], 'ZoneType_t
↪']
#>>      ...
C.convertPyTree2File(t, 'out.cgns')

```

Converter.Internal.**groupBCByBCType**(t, btype='BCWall', name='FamWall')

For each base, gather all BCs of same BCType in a family named FamilyName. The family node is created and the BC is tagged with BC family.

Exists also as in place version (\_groupBCByBCType) that modifies t and returns None.

#### Parameters

- **t** (pyTree or list of Base nodes) – input data
- **btype** (string) – Type of BC to be grouped in a family
- **name** (string) – name of family to be created

**Returns** reference copy of input

**Return type** same as input

*Example of use:*

- Group BC from their types (pyTree):

```

# - groupBCByType (PyTree) -
import Converter.PyTree as C
import Converter.Internal as Internal
import Generator.PyTree as G

a = G.cart((0.,0.,0.), (0.1,0.1,0.1), (10,10,10))
a = C.fillEmptyBCWith(a, 'wall', 'BCWall')

t = C.newPyTree(['Base',a])
Internal._groupBCByBCType(t, 'BCWall', 'FamWall')

#>> ['CGNSTree',None,[2 sons], 'CGNSTree_t']
#>> |_['CGNSLibraryVersion',array([3.1],dtype='float64'),[0 son],
↪'CGNSLibraryVersion_t']
#>> |_['Base',array(shape=(2,),dtype='int32',order='F'),[2 sons], 'CGNSBase_t']
#>> |_['cart',array(shape=(3, 3),dtype='int32',order='F'),[3 sons], 'Zone_t
↪']

```

```

#>> | ...
#>> | | _['ZoneBC',None,[6 sons],'ZoneBC_t']
#>> | | | _['wall1',array('FamilySpecified',dtype='|S1'),[2 sons],'BC_t
↳']
#>> | | | | _['PointRange',array(shape=(3, 2),dtype='int32',order='F
↳'),[0 son],'IndexRange_t']
#>> | | | | _['FamilyName',array('FamWall',dtype='|S1'),[0 son],
↳'FamilyName_t']
#>> | | | | _['wall2',array('FamilySpecified',dtype='|S1'),[2 sons],'BC_t
↳']
#>> | | | | | _['PointRange',array(shape=(3, 2),dtype='int32',order='F
↳'),[0 son],'IndexRange_t']
#>> | | ...
#>> | | _['FamWall',None,[1 son],'Family_t']
#>> | | | _['FamilyBC',array('BCWall',dtype='|S1'),[0 son],'FamilyBC_t']
C.convertPyTree2File(t, 'out.cgns')

```

---

**Note:** New in version 2.4

---

## 3.8 Create specific CGNS nodes

`Converter.Internal.newCGNSTree()`

Create a tree node with the CGNS version node.

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new tree node (pyTree):

```

# - newCGNSTree (pyTree) -
import Converter.Internal as Internal
t = Internal.newCGNSTree(); print(t)
#>> ['CGNSTree', None, [['CGNSLibraryVersion', array([ 3.1]), []],
↳'CGNSLibraryVersion_t']], 'CGNSTree_t']

```

`Converter.Internal.newCGNSBase(name='Base', cellDim=3, physDim=3, parent=None)`

Create a base node. `cellDim` is the dimension of zone cells of this base, `physDim` is the dimension of physical space. For example, triangle zones in 3D space will correspond to `cellDim=2` and `physDim=3`. If `parent` is not `None`, attach it to parent node.

**Parameters**

- **name** (string) – name of base
- **cellDim** (int) – dimension of zone cells in Base (1, 2, 3)
- **physDim** (int) – physical dimension of space (1, 2, 3)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new base node (pyTree):

```
# - newCGNSBase (pyTree) -
import Converter.Internal as Internal

# Create a base node
b = Internal.newCGNSBase('Base'); print(b)
#>> ['Base', array([3, 3], dtype=int32), [], 'CGNSBase_t']

# Create a base node and attach it to tree
t = Internal.newCGNSTree()
Internal.newCGNSBase('Base', 3, 3, parent=t); Internal.printTree(t)
#>> ['CGNSTree',None,[2 sons],'CGNSTree_t']
#>>  |['_['CGNSLibraryVersion',array([3.1],dtype='float64'),[0 son],
↪'_CGNSLibraryVersion_t']
#>>  |['_['Base',array(shape=(2,),dtype='int32',order='F'),[0 son],'CGNSBase_t']
```

---

`Converter.Internal.newZone(name='Zone', zsize=None, ztype='Structured', family=None, parent=None)`

Create a zone node. `zsize` is the dimension of zone, `ztype` is the type of zone ('Structured' or 'Unstructured'). If family is not None, attach a zone family node. If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of zone
- **zsize** (list of integers) – number of points, number of elements, 0
- **ztype** (string) – 'Structured' or 'Unstructured'
- **family** (string) – optional family name
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new zone node (pyTree):

```
# - newZone (pyTree) -
import Converter.Internal as Internal

# Create a zone node
z = Internal.newZone('Zone', zsize=[[10, 2, 0]], ztype='Structured'); Internal.
↳printTree(z)
#>> ['Zone',array(shape=(3, 1),dtype='int32',order='F'),[1 son],'Zone_t']
#>>  |['_ZoneType',array('Structured',dtype='|S1'),[0 son],'ZoneType_t']

# Create a zone node and attach it to tree
t = Internal.newCGNSTree()
b = Internal.newCGNSBase('Base', 3, 3, parent=t)
z = Internal.newZone('Zone', [[10],[2],[0]], 'Structured', parent=b)
```

`Converter.Internal.newGridCoordinates(name=Internal.__GridCoordinates__,  
parent=None)`

Create a GridCoordinates node. If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of GridCoordinates container
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GridCoordinates node (pyTree):

```
# - newGridCoordinates (pyTree) -
import Converter.Internal as Internal

# Create a GridCoordinates node
n = Internal.newGridCoordinates(); print(n)
#>> ['GridCoordinates', None, [], 'GridCoordinates_t']

# Create a zone node
z = Internal.newZone('Zone', zsize=[[10],[2],[0]], ztype='Structured')
n = Internal.newGridCoordinates(parent=z); Internal.printTree(z)
```

```
#>> ['Zone',array(shape=(3, 1),dtype='int32',order='F'),[2 sons],'Zone_t']
#>>  |['_['ZoneType',array('Structured',dtype='|S1'),[0 son],'ZoneType_t']
#>>  |['_['GridCoordinates',None,[0 son],'GridCoordinates_t']
```

---

`Converter.Internal.newDataArray(name='Data', value=None, parent=None)`

Create a DataArray node. `value` can be a string, an int, a float, a numpy of ints, a numpy of floats. If `parent` is not `None`, attach it to parent node.

#### Parameters

- **name** (string) – name of node
- **value** (string, int, float, numpy) – value to put in node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new DataArray node (pyTree):

```
# - newDataArray (pyTree) -
import Converter.Internal as Internal
import numpy

# Create a DataArray node
n = Internal.newDataArray('CoordinateX', numpy.zeros(10)); print(n)
#>> ['CoordinateX', array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
↳ [], 'DataArray_t']

# Attach it to a parent node
g = Internal.newGridCoordinates()
Internal.newDataArray('CoordinateX', value=numpy.arange(0,10), parent=g)
Internal.newDataArray('CoordinateY', value=numpy.zeros(10), parent=g)
Internal.newDataArray('CoordinateZ', value=numpy.zeros(10), parent=g); Internal.
↳ printTree(g)
#>> ['GridCoordinates',None,[3 sons],'GridCoordinates_t']
#>>  |['_['CoordinateX',array(shape=(10,),dtype='int64',order='F'),[0 son],
↳ 'DataArray_t']
#>>  |['_['CoordinateY',array(shape=(10,),dtype='float64',order='F'),[0 son],
↳ 'DataArray_t']
#>>  |['_['CoordinateZ',array(shape=(10,),dtype='float64',order='F'),[0 son],
↳ 'DataArray_t']
```

---

`Converter.Internal.newDataClass(value='UserDefined', parent=None)`  
 Create a DataClass node. If parent is not None, attach it to parent node.

#### Parameters

- **value** (string) – value to put in node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new DataClass node (pyTree):

```
# - newDataClass (pyTree) -
import Converter.Internal as Internal

# Create a DataClass node
n = Internal.newDataClass('Dimensional'); Internal.printTree(n)
#>> ['DataClass',array('Dimensional',dtype='|S1'),[0 son],'DataClass_t']

# Attach it to a parent node
d = Internal.newDiscreteData('DiscreteData')
Internal.newDataClass('Dimensional', parent=d)
```

`Converter.Internal.newDimensionalUnits(massUnit='Kilogram', lengthUnit='Meter', timeUnit='Second', temperatureUnit='Kelvin', angleUnit='Radian', parent=None)`

Create a DimensionalUnits node. Arguments describe the units of the problem. If parent is not None, attach it to parent node.

#### Parameters

- **massUnit** (string in 'Null', 'UserDefined', 'Kilogram', 'Gram', 'Slug', 'PoundMass',) – mass unit you want
- **lengthUnit** (string in 'Null', 'UserDefined', 'Meter', 'Centimeter', 'Millimeter', 'Foot', 'Inch') – length unit you want
- **timeUnit** (string in 'Null', 'UserDefined', 'Second') – time unit you want
- **temperatureUnit** (string in 'Null', 'UserDefined', 'Kelvin', 'Celsius', 'Rankine') – temprature unit you want

- **angleUnit** (string in 'Null', 'UserDefined', 'Radian', 'Degree') – angle unit you want
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new DimensionalUnits node (pyTree):

```
# - newDimensionalUnits (pyTree) -
import Converter.Internal as Internal
import numpy

# Create a DimensionalUnits node
n = Internal.newDimensionalUnits(massUnit='Kilogram')
Internal.printTree(n)
#>> ['DimensionalUnits',array(shape=(32, 5),dtype='|S1',order='F'),[0 son],
↪'DimensionalUnit_t']

# Attach it to a parent node
d = Internal.newDataArray('CoordinateX', numpy.zeros(10))
Internal.newDimensionalUnits(lengthUnit='Meter', parent=d)
```

---

`Converter.Internal.newDimensionalExponents`(*massExponent*='Kilogram', *lengthExponent*='Meter', *timeExponent*='Second', *temperatureExponent*='Kelvin', *angleExponent*='Radian', *parent*=None)

Create a DimensionalExponents node. Arguments describe the unit exponent of the problem. If parent is not None, attach it to parent node.

**Parameters**

- **massExponent** (float) – exponent for mass
- **lengthExponent** (float) – exponent for length
- **timeExponent** (float) – exponent for time
- **temperatureExponent** (float) – exponent for temperature
- **angleExponent** (float) – exponent for angle
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node



*Example of use:*

- Create a new DimensionalExponents node (pyTree):

```
# - newDimensionalExponents (pyTree) -
import Converter.Internal as Internal

# Create a DimensionalExponents node
# Example for velocity exponents (m/s)
n = Internal.newDimensionalExponents(massExponent=0., lengthExponent=1.,
↳timeExponent=-1., temperatureExponent=0., angleExponent=0.); Internal.
↳printTree(n)
#>> ['DimensionalExponents',array(shape=(5,),dtype='float64',order='F'),[0 son],
↳'DimensionalExponents_t']

# Attach it to a parent node
d = Internal.newGridCoordinates()
Internal.newDataClass('Dimensional', parent=d)
Internal.newDimensionalExponents(massExponent=0., lengthExponent=1.,
↳timeExponent=0., temperatureExponent=0., angleExponent=0., parent=d)
```

`Converter.Internal.newDataConversion`(*conversionScale=1.*, *conversionOffset=0.*,  
*parent=None*)

Create a DataConversion node. Arguments describe the conversion factors.  
Data(raw) = Data(nondimensional)\*ConversionScale + ConversionOffset If parent  
is not None, attach it to parent node.

#### Parameters

- **conversionScale** (float) – scale for conversion
- **conversionOffset** – offset for conversion
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new DataConversion node (pyTree):

```
# - newDataConversion (pyTree) -
import Converter.Internal as Internal

# Data(raw) = Data(nondimensional)*ConversionScale + ConversionOffset

# Create a DataConversion node
n = Internal.newDataConversion(conversionScale=1., conversionOffset=0.);
↳Internal.printTree(n)
```

```
#>> ['DataConversion', array(shape=(2,), dtype='float64', order='F'), [0 son],
↪ 'DataConversion_t']

# Attach it to a parent node
d = Internal.newDiscreteData('DiscreteData')
Internal.newDataClass('NondimensionalParameter', parent=d)
Internal.newDataConversion(conversionScale=1., conversionOffset=0., parent=d)
```

`Converter.Internal.newDescriptor`(*name*='Descriptor', *value*="", *parent*=None)  
Create a Descriptor node. If parent is not None, attach it to parent node.

### Parameters

- **name** (string) – name of node
- **value** (string) – value to put in node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new Descriptor node (pyTree):

```
# - newDescriptor (pyTree) -
import Converter.Internal as Internal

# Create a Descriptor node
n = Internal.newDescriptor(name='Descriptor', value='Mesh exported from_
↪Cassiopee 2.2'); print(n)
#>> ['Descriptor', array(..), [], 'Descriptor_t']

# Attach it to a parent node
b = Internal.newCGNSBase('Base')
Internal.newDescriptor('Descriptor', 'Aircraft with nacelle Mach=0.7 Re=4.
↪million alpha=-2', parent=b)
```

`Converter.Internal.newGridLocation`(*value*='CellCenter', *parent*=None)  
Create a GridLocation node. If parent is not None, attach it to parent node.

### Parameters

- **value** (string) – value to put in node ('CellCenter', 'Vertex'...)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GridLocation node (pyTree):

```
# - newGridLocation (pyTree) -
import Converter.Internal as Internal

# Create a GridLocation node
n = Internal.newGridLocation(value='CellCenter'); Internal.printTree(n)
#>> ['GridLocation',array('CellCenter',dtype='|S1'),[0 son],'GridLocation_t']

# Attach it to a parent node
d = Internal.newBC('wall', [1,80,30,30,1,2], 'BCWall')
Internal.newGridLocation('Vertex', parent=d)
```

`Converter.Internal.newIndexArray(name='Index', value=None, parent=None)`  
 Create a indexArray node. If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of node
- **value** (list of integers) – integer values of indices
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new IndexArray node (pyTree):

```
# - newIndexArray (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newIndexArray(name='Index', value=[101,51,22,1024,78]); Internal.
↪printTree(n)
#>> ['Index',array(shape=(5,),dtype='int32',order='F'),[0 son],'IndexArray_t']

# Attach it to a parent node
d = Internal.newFlowSolution('FlowSolution', 'Vertex')
Internal.newIndexArray('Index', [101,51,22,1024,78], parent=d)
```

`Converter.Internal.newPointList`(*name='PointList', value=None, parent=None*)  
Create a PointList node. If parent is not None, attach it to parent node.

### Parameters

- **name** (string) – name of node
- **value** (list of integers) – list of point indices
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new PointList node (pyTree):

```
# - newPointList (pyTree) -
import Converter.Internal as Internal

# Create a Descriptor node
n = Internal.newPointList(name='PointList', value=[101,51,22,1036,2]); Internal.
↳ printTree(n)
#>> ['PointList',array(shape=(5,),dtype='int32',order='F'),[0 son],'IndexArray_t
↳ ']

# Attach it to a parent node
d = Internal.newBC('wall', [1,80,30,30,1,2], 'BCWall')
Internal.newPointList('PointList', [101,51,22,1036,2], parent=d)
```

`Converter.Internal.newPointRange`(*name='PointRange', value=None, parent=None*)

Create a PointRange node. If parent is not None, attach it to parent node.

### Parameters

- **name** (string) – name of node
- **value** (list of integers) – list of point indices ([imin,imax,jmin,jmax,kmin,kmax] for a structured point range)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new PointRange node (pyTree):

```
# - newPointRange (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newPointRange(name='PointRange', value=[1,10,1,10,5,5]); Internal.
↳ printTree(n)
#>> ['PointRange',array(shape=(3, 2),dtype='int32',order='F'),[0 son],
↳ 'IndexRange_t']

# Attach it to a parent node
d = Internal.newBC('wall', [1,80,30,30,1,2], 'BCWall')
Internal.newPointRange('PointRange', [1,71,29,29,1,5], parent=d)
```

Converter.Internal.**newRind**(value=None, parent=None)

Create a Rind node. Rind contains the number of ghost cells for a structured block. If parent is not None, attach it to parent node.

#### Parameters

- **value** (list of integers) – list of integers (6 for a structured block)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new Rind node (pyTree):

```
# - newRind (pyTree) -
import Converter.Internal as Internal

# Create a Rind node (Rind contains the number of ghost cells for a structured_
↳ block)
n = Internal.newRind([0,0,0,0,1,1]); Internal.printTree(n)
#>> ['Rind',array(shape=(6,),dtype='int32',order='F'),[0 son],'Rind_t']

# Attach it to a parent node
d = Internal.newGridCoordinates()
Internal.newRind([0,0,0,0,1,1], parent=d)
```

Converter.Internal.**newSimulationType**(value='TimeAccurate', parent=None)

Create a SimulationType node. If parent is not None, attach it to parent node.

#### Parameters

- **value** (string) – value of the node ('TimeAccurate','NonTimeAccurate')
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new SimulationType node (pyTree):

```
# - newSimulationType (pyTree) -
import Converter.Internal as Internal

# Create a simulation type node
b = Internal.newSimulationType(value='NonTimeAccurate'); Internal.printTree(b)
#>> ['SimulationType',array('NonTimeAccurate',dtype='|S1'),[0 son],
↳ 'SimulationType_t']

# Attach it to parent
b = Internal.newCGNSBase('Base', 3, 3)
Internal.newSimulationType('TimeAccurate', parent=b)
```

---

Converter.Internal.**newOrdinal**(value=0, parent=None)

Create an Ordinal node. If parent is not None, attach it to parent node.

**Parameters**

- **value** (integer) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new Ordinal node (pyTree):

```
# - newOrdinal (pyTree) -
import Converter.Internal as Internal

# Create a ordinal node
b = Internal.newOrdinal(value=1); Internal.printTree(b)
#>> ['Ordinal',array([1],dtype='int32'),[0 son], 'Ordinal_t']

# Attach it to zone
```

```
z = Internal.newZone('Zone', [[10],[2],[0]], 'Structured')
Internal.newOrdinal(value=1, parent=z)
```

Converter.Internal.**newDiscreteData**(name='DiscreteData', parent=None)

Create a DiscreteData node. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new DiscreteData node (pyTree):

```
# - newDiscreteData (pyTree) -
import Converter.Internal as Internal

# Create a discrete data node
b = Internal.newDiscreteData(name='DiscreteData'); Internal.printTree(b)
#>> ['DiscreteData',None,[0 son],'DiscreteData_t']

# Attach it to zone
z = Internal.newZone('Zone', [[10],[2],[0]], 'Structured')
Internal.newDiscreteData('DiscreteData', parent=z)
```

Converter.Internal.**newIntegralData**(name='IntegralData', parent=None)

Create an IntegralData node. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new IntegralData node (pyTree):

```
# - newIntegralData (pyTree) -
import Converter.Internal as Internal

# Create an integral data node
n = Internal.newIntegralData(name='IntegralData'); Internal.printTree(n)
#>> ['IntegralData',None,[0 son],'IntegralData_t']

# Attach it to base
b = Internal.newCGNSBase('Base', 3, 3)
Internal.newIntegralData('IntegralData', parent=b)
```

`Converter.Internal.newElements`(*name='Elements', etype='UserDefined', econnectivity=None, erange=None, eboundary=0, parent=None*)

Create a Elements node. *etype* is the element type ('BAR', 'TRI', 'QUAD', ...), *econnectivity* is the connectivity numpy array and *eboundary* ... If *parent* is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **etype** (string) – type of the elements ('BAR', 'TRI', 'QUAD', 'TETRA', 'HEXA', 'PENTA', 'NODE', 'PYRA', 'NGON', 'NFACE')
- **econnectivity** (numpy array) – connectivity of elements
- **erange** (list of two integers) – element range
- **eboundary** (integer) – number of elements at boundary (0 by default)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new Elements node (pyTree):

```
# - newElements (pyTree) -
import Converter.Internal as Internal

# Create an elements node
b = Internal.newElements(name='Elements', etype='UserDefined',
↪econnectivity=None, eboundary=0); Internal.printTree(b)
#>> ['Elements',array(shape=(2,),dtype='int32',order='F'),[2 sons],'Element_t']
#>> |['_ElementConnectivity',None,[0 son],'DataArray_t']
```



```
#>>  |_[ 'ElementRange',None,[0 son], 'IndexRange_t']

# Attach it to zone
z = Internal.newZone('Zone', [[10],[2],[0]], 'Unstructured')
Internal.newElements(name='Elements', etype='UserDefined', econnectivity=None,
↳eboundary=0, parent=z)
```

Converter.Internal.**newParentElements**(value=None, parent=None)

Create a ParentElements node. value is a numpy array. If parent is not None, attach it to parent node.

#### Parameters

- **value** (numpy array) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ParentElements node (pyTree):

```
# - newParentElements (pyTree) -
import Converter.Internal as Internal

# Create a parent elements node
n = Internal.newParentElements(value=None); Internal.printTree(n)
#>> ['ParentElements',None,[0 son], 'DataArray_t']

# Attach it to elements
d = Internal.newElements('Elements', 'UserDefined', None, 0)
Internal.newParentElements(None, parent=d)
```

Converter.Internal.**newParentElementsPosition**(value=None, parent=None)

Create a ParentElementsPosition node. value is a numpy array. If parent is not None, attach it to parent node.

#### Parameters

- **value** (numpy array) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ParentElementsPosition node (pyTree):

```
# - newParentElementsPosition (pyTree) -
import Converter.Internal as Internal

# Create a parent elements position node
n = Internal.newParentElementsPosition(value=None); Internal.printTree(n)
#>> ['ParentElementsPosition',None,[0 son],'DataArray_t']

# Attach it to elements
d = Internal.newElements(name='Elements', etype='UserDefined',
↪econnectivity=None, eboundary=0)
Internal.newParentElementsPosition(None, parent=d)
```

---

Converter.Internal.newZoneBC(parent=None)

Create a ZoneBC node. If parent is not None, attach it to parent node.

**Parameters** parent (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ZoneBC node (pyTree):

```
# - newZoneBC (pyTree) -
import Converter.Internal as Internal

# Create a zoneBC node
n = Internal.newZoneBC(); Internal.printTree(n)
#>> ['ZoneBC',None,[0 son],'ZoneBC_t']

# Attach it to a parent node
z = Internal.newZone('Zone', zsize=[[10],[2],[0]], ztype='Structured')
Internal.newZoneBC(parent=z)
```

---

Converter.Internal.newBC(name='BC', pointRange=None, pointList=None,  
btype='Null', family=None, parent=None)

Create a BC node. It can be defined by a pointRange for structured zones or a pointList of faces for unstructured zones. btype specifies the BC type. A BC family-Name can also be defined. If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of the node
- **pointRange** (list of integers) – list of point indices ([imin,imax,jmin,jmax,kmin,kmax] for a structured point range)
- **pointList** (list of integers) – list of point indices (for unstructured grid)
- **btype** (string) – type of BC (BCWall, BCFarfield, FamilySpecified...)
- **family** (string) – name of the family for a family specified BC
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new BC node (pyTree):

```
# - newBC (pyTree) -
import Converter.Internal as Internal

# Create a BC node
n = Internal.newBC(name='BC', pointList=[22,1036,101,43], btype='BCFarfield')
Internal.printTree(n)
#>> ['BC',array('BCFarfield',dtype='|S1'),[1 son],'BC_t']
#>>  |['_PointList',array(shape=(4,),dtype='int32',order='F'),[0 son],
↪ 'IndexArray_t']

# Attach it to a parent node
d = Internal.newZoneBC()
Internal.newBC('BC', [1,45,1,21,1,1], 'BCWall', parent=d)
```

`Converter.Internal.newBCDataSet(name='BCDataSet', value='Null', gridLocation=None, parent=None)`

Create a BCDataSet node. value must be a BCType. GridLocation ('FaceCenter', 'Vertex') can be specified. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **value** (string) – value of node (UserDefined, BCWall...)
- **gridLocation** (string) – location of the grid points (Vertex, FaceCenter, CellCenter...)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new BCDataSet node (pyTree):

```
# - newBCDataSet (pyTree) -
import Converter.Internal as Internal

# Create a BC data set node
n = Internal.newBCDataSet(name='BCDataSet', value='BCWall'); Internal.
↪ printTree(n)
#>> ['BCDataSet',array('BCWall',dtype='|S1'),[0 son],'BCDataSet_t']

# Attach it to a parent node
d = Internal.newBC(name='BC', pointList=[22,1036,101,43], btype='BCFarfield')
Internal.newBCDataSet(name='BCDataSet', value='UserDefined', parent=d)

# Complete BC + BCDataSet + BCData
b = Internal.newBC(name='BC', pointList=[22,1036,101,43], btype='BCFarfield')
d = Internal.newBCDataSet(name='BCDataSet', value='UserDefined', gridLocation=
↪ 'FaceCenter', parent=b)
d = Internal.newBCData('BCNeumann', parent=d)
d = Internal.newDataArray('Density', value=[1.,1.,1.,1.], parent=d)
```

---

Converter.Internal.**newBCData**(name='BCData', parent=None)

Create a BCData node. If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new BCData node (pyTree):

```
# - newBCData (pyTree) -
import Converter.Internal as Internal

# Create a BC data node
n = Internal.newBCData(name='BCData'); Internal.printTree(n)
#>> ['BCData',None,[0 son],'BCData_t']
```

```
# Attach it to a parent node
d = Internal.newBCDataSet(name='BCDataSet', value='UserDefined')
Internal.newBCData('BCNeumann', parent=d); Internal.printTree(d)
#>> ['BCDataSet',array('UserDefined',dtype='|S1'),[1 son],'BCDataSet_t']
#>>  |['_BCNeumann',None,[0 son],'BCData_t']
```

`Converter.Internal.newBCProperty(wallFunction='Null', area='Null', parent=None)`

Create a BCProperty node. If parent is not None, attach it to parent node.

#### Parameters

- **wallFunction** (string) – type of wall function (Null, UserDefined, Generic)
- **area** (string) – type of area (Null, UserDefined, BleedArea, CaptureArea)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new BCProperty node (pyTree):

```
# - newBCProperty (pyTree) -
import Converter.Internal as Internal

# Create a BC property node
n = Internal.newBCProperty(wallFunction='Null', area='Null'); Internal.
↪printTree(n)
#>> ['BCProperty',None,[2 sons],'BCProperty_t']
#>>  |['_WallFunctionType',array('Null',dtype='|S1'),[0 son],
↪'_WallFunctionType_t']
#>>  |['_Area',array('Null',dtype='|S1'),[0 son],'Area_t']

# Attach it to a parent node
d = Internal.newBC(name='BC', pointList=[22,1036,101,43], btype='BCWall')
Internal.newBCProperty(wallFunction='Null', area='Null', parent=d)
```

`Converter.Internal.newAxisSymmetry(referencePoint=[0.,0.,0.], axisVector=[0.,0.,0.], parent=None)`

Create a AxisSymmetry node. If parent is not None, attach it to parent node.

**Parameters**

- **referencePoint** (list of 3 floats) – coordinates of the axis point
- **axisVector** (list of 3 floats) – axis vector
- **parent** (pyTree node) – optional parent node

**Returns** created node**Return type** pyTree node*Example of use:*

- Create a new AxiSymmetry node (pyTree):

```
# - newAxiSymmetry (pyTree) -
import Converter.Internal as Internal

# Create an axisymmetry node
n = Internal.newAxiSymmetry(referencePoint=[0.,0.,0.], axisVector=[0.,0.,0.]);
↳Internal.printTree(n)
#>> ['AxiSymmetry',None,[2 sons],'AxiSymmetry_t']
#>>   |['_AxiSymmetryReferencePoint',array(shape=(3,),dtype='float64',order='F')
↳),[0 son],'DataArray_t']
#>>   |['_AxiSymmetryAxisVector',array(shape=(3,),dtype='float64',order='F'),
↳[0 son],'DataArray_t']

# Attach it to base
b = Internal.newCGNSBase('Base', 3, 3)
Internal.newAxiSymmetry([0.,0.,0.], [0.,0.,0.], parent=b)
```

---

```
Converter.Internal.newRotatingCoordinates(rotationCenter=[0.,0.,0.], rotationRateVector=[0.,0.,0.], parent=None)
```

Create a RotatingCoordinates node. If parent is not None, attach it to parent node.

**Parameters**

- **rotationCenter** (list of 3 floats) – coordinates of the rotation center
- **rotationRateVector** (list of 3 floats) – vector of rotation rate
- **parent** (pyTree node) – optional parent node

**Returns** created node**Return type** pyTree node*Example of use:*

- Create a new RotatingCoordinates node (pyTree):

```
# - newRotatingCoordinates (pyTree) -
import Converter.Internal as Internal

# Create an rotating coordinates node
n = Internal.newRotatingCoordinates(rotationCenter=[0.,0.,0.],
↳ rotationRateVector=[0.,0.,0.]); Internal.printTree(n)
#>> ['RotatingCoordinates',None,[2 sons],'RotatingCoordinates_t']
#>>   |['_RotationCenter',array(shape=(3,),dtype='float64',order='F'),[0 son],
↳ 'DataArray_t']
#>>   |['_RotationRateVector',array(shape=(3,),dtype='float64',order='F'),[0.
↳ son],'DataArray_t']

# Attach it to base
b = Internal.newCGNSBase('Base', 3, 3)
Internal.newRotatingCoordinates([0.,0.,0.], [0.,0.,0.], parent=b)
```

`Converter.Internal.newFlowSolution(name=__FlowSolutionNodes__, gridLocation='Vertex', parent=None)`

Create a newFlowSolution node. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string or container) – name or the container of the flow solution node
- **gridLocation** (string (Vertex, CellCenter...)) – location of the node grid points
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new FlowSolution node (pyTree):

```
# - newFlowSolution (pyTree) -
import Converter.Internal as Internal

# Create a flow solution node
n = Internal.newFlowSolution(name='FlowSolution', gridLocation='Vertex');
↳ Internal.printTree(n)
#>> ['FlowSolution',None,[1 son],'FlowSolution_t']
#>>   |['_GridLocation',array('Vertex',dtype='|S1'),[0 son],'GridLocation_t']
```

```
# Attach it to a parent node
z = Internal.newZone('Zone', zsize=[[10],[2],[0]], ztype='Structured')
Internal.newFlowSolution(name='FlowSolution', gridLocation='Vertex', parent=z)
```

---

`Converter.Internal.newZoneGridConnectivity`(name='ZoneGridConnectivity', parent=None)

Create a newZoneGridConnectivity node. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ZoneGridConnectivity node (pyTree):

```
# - newZoneGridConnectivity (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newZoneGridConnectivity(name='ZoneGridConnectivity'); Internal.
↳ printTree(n)
#>> ['ZoneGridConnectivity',None,[0 son],'ZoneGridConnectivity_t']

# Attach it to a parent node
z = Internal.newZone('Zone', zsize=[[10],[2],[0]], ztype='Structured')
Internal.newZoneGridConnectivity('ZoneGridConnectivity', parent=z)
```

---

`Converter.Internal.newGridConnectivity1to1`(name='Match', donor-Name=None, pointRange=None, pointList=None, pointRange-Donor=None, pointList-Donor=None, transform=None, parent=None)

Create a newGridConnectivity1to1 node. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node



- **donorName** (string) – name of donor zone
- **pointRange** (list of integers) – list of point indices of the local zone ([imin,imax,jmin,jmax,kmin,kmax] for a structured point range)
- **pointList** (list of integers) – list of point indices of the local zone (for unstructured grid)
- **pointRangeDonor** (list of integers) – list of point indices of the donor zone ([imin,imax,jmin,jmax,kmin,kmax] for a structured point range)
- **pointListDonor** (list of integers) – list of point indices of the donor zone (for unstructured grid)
- **transform** (list of integers ([1,2,3], [-2,-1,-3]..)) – transformation of the orientation between local and donor zones
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GridConnectivity1to1 node (pyTree):

```
# - newGridConnectivity1to1 (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newGridConnectivity1to1(name='Match', donorName='blk1',
↳pointRange=[1,1,1,33,1,69], pointRangeDonor=[51,51,1,33,1,69], transform=[1,2,
↳3]); Internal.printTree(n)
#>> ['Match',array('blk1',dtype='|S1'),[2 sons],'GridConnectivity1to1_t']
#>>   |['_PointRange',array(shape=(3, 2),dtype='int32',order='F'),[0 son],
↳'_IndexRange_t']
#>>   |['_PointRangeDonor',array(shape=(3, 2),dtype='int32',order='F'),[0 son],
↳'_IndexRange_t']
#>>   |['_Transform',array(shape=(3,),dtype='int32',order='F'),[0 son],
↳'int[IndexDimension]"]']

# Attach it to a parent node
d = Internal.newZoneGridConnectivity(name='ZoneGridConnectivity')
Internal.newGridConnectivity1to1(name='Match', donorName='blk1', pointRange=[1,
↳1,1,33,1,69], pointRangeDonor=[51,51,1,33,1,69], transform=None, parent=d)
```

`Converter.Internal.newGridConnectivity(name='Overlap', donorName=None, ctype='Overset', parent=None)`

Create a newGridConnectivity node. If parent is not None, attach it to parent node.

### Parameters

- **name** (string) – name of the node
- **donorName** (string) – name of donor zone
- **ctype** (string) – connectivity type ('Overset')
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GridConnectivity node (pyTree):

```
# - newGridConnectivity (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newGridConnectivity(name='Match', donorName='blk1', ctype=
↳ 'Abutting1to1'); Internal.printTree(n)
#>> ['Match',array('blk1',dtype='|S1'),[1 son], 'GridConnectivity_t']
#>>   |['_GridConnectivityType',array('Abutting1to1',dtype='|S1'),[0 son],
↳ 'GridConnectivityType_t']

# Attach it to a parent node
d = Internal.newZoneGridConnectivity(name='ZoneGridConnectivity')
Internal.newGridConnectivity(name='Match', donorName='blk1', ctype='Abutting1to1
↳ ', parent=d)
```

`Converter.Internal.newGridConnectivityType(ctype='Overset', parent=None)`

Create a newGridConnectivityType node. If parent is not None, attach it to parent node.

### Parameters

- **ctype** (string) – connectivity type ('Overset')
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GridConnectivityType node (pyTree):

```
# - newGridConnectivityType (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newGridConnectivityType(ctype='Abutting1to1'); Internal.
↳ printTree(n)
#>> ['GridConnectivityType',array('Abutting1to1',dtype='|S1'),[0 son],
↳ 'GridConnectivityType_t']

# Attach it to a parent node
d = Internal.newGridConnectivity(name='Match', donorName='blk1', ctype=None)
Internal.newGridConnectivityType(ctype='Abutting1to1', parent=d)
```

Converter.Internal.newGridConnectivityProperty(parent=None)

Create a newGridConnectivityProperty node. If parent is not None, attach it to parent node.

**Parameters** parent (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GridConnectivityProperty node (pyTree):

```
# - newGridConnectivityProperty (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newGridConnectivityProperty(); Internal.printTree(n)
#>> ['GridConnectivityProperty',None,[0 son],'GridConnectivityProperty_t']

# Attach it to a parent node
d = Internal.newGridConnectivity(name='Match', donorName='blk1', ctype=
↳ 'Abutting1to1')
Internal.newGridConnectivityProperty(parent=d)
```

Converter.Internal.newPeriodic(rotationCenter=[0.,0.,0.], rotationAn-  
 gle=[0.,0.,0.], translation=[0.,0.,0.], par-  
 ent=None)

Create a Periodic node. If parent is not None, attach it to parent node.

**Parameters**

- **rotationCenter** (list of 3 floats) – coordinates of the rotation center
- **rotationAngle** (list of 3 floats) – angles of rotation
- **translation** (list of 3 floats) – translation vector
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new Periodic node (pyTree):

```
# - newPeriodic (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newPeriodic(rotationCenter=[0.,0.,0.], rotationAngle=[0.,0.,0.],
↳ translation=[0.,0.,0.]); Internal.printTree(n)
#>> ['Periodic',None,[3 sons],'Periodic_t']
#>>   |['_RotationCenter',array(shape=(3,),dtype='float64',order='F'),[0 son],
↳ 'DataArray_t']
#>>   |['_RotationAngle',array(shape=(3,),dtype='float64',order='F'),[0 son],
↳ 'DataArray_t']
#>>   |['_Translation',array(shape=(3,),dtype='float64',order='F'),[0 son],
↳ 'DataArray_t']

# Attach it to a parent node
d = Internal.newGridConnectivityProperty()
Internal.newPeriodic(rotationCenter=[0.,0.,0.], rotationAngle=[0.,0.,0.],
↳ translation=[0.,0.,0.], parent=d)
```

---

`Converter.Internal.newOversetHoles(name='OversetHoles', pointRange=None, pointList=None, parent=None)`  
Create a OversetHoles node. If parent is not None, attach it to parent node.

#### Parameters

- **name** (list of 3 floats) – coordinates of the rotation center
- **pointRange** (list of integers) – list of point indices ([imin,imax,jmin,jmax,kmin,kmax] for a structured point range)
- **pointList** (list of integers) – list of point indices (for unstructured grid)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new OversetHoles node (pyTree):

```
# - newOversetHoles (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newOversetHoles(name='OversetHoles', pointRange=None,
↪pointList=None); Internal.printTree(n)
#>> ['OversetHoles',None,[0 son],'OversetHoles_t']

# Attach it to a parent node
d = Internal.newZoneGridConnectivity(name='ZoneGridConnectivity')
Internal.newOversetHoles(name='OversetHoles', pointList=[22,1036,101,43],
↪parent=d)
```

`Converter.Internal.newFlowEquationSet(parent=None)`

Create a FlowEquationSet node. If parent is not None, attach it to parent node.

**Parameters** parent (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new FlowEquationSet node (pyTree):

```
# - newFlowEquationSet (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newFlowEquationSet(); Internal.printTree(n)
#>> ['FlowEquationSet',None,[0 son],'FlowEquationSet_t']

# Create a node and attach it to parent
t = Internal.newCGNSTree()
b = Internal.newCGNSBase('Base', 3, 3, parent=t)
n = Internal.newFlowEquationSet(parent=b)
```

`Converter.Internal.newGoverningEquations(value='Euler', parent=None)`

Create a GoverningEquations node. value is the equation type in 'FullPotential',

'Euler', 'NSLaminar', 'NSTurbulent', 'NSLaminarIncompressible', 'NSTurbulentIncompressible'. If parent is not None, attach it to parent node.

**Parameters**

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GoverningEquations node (pyTree):

```
# - newGoverningEquation (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newGoverningEquations(value='Euler'); Internal.printTree(n)
#>> ['GoverningEquations', array('Euler', dtype='|S1'), [0 son],
↪ 'GoverningEquations_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newGoverningEquations(value='NSTurbulent', parent=t)
```

---

Converter.Internal.**newGasModel**(value='Ideal', parent=None)

Create a GasModel node. value is the model type in 'Ideal', 'VanderWaals', 'CaloricallyPerfect', 'ThermallyPerfect', 'ConstantDensity', 'RedlichKwong'. If parent is not None, attach it to parent node.

**Parameters**

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GasModel node (pyTree):

```
# - newGasModel (pyTree) -
import Converter.Internal as Internal

# Create a node
```

```

z = Internal.newGasModel(value='Ideal'); Internal.printTree(z)
#>> ['GasModel',array('Ideal',dtype='|S1'),[0 son],'GasModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newGasModel(value='Ideal', parent=t)

```

Converter.Internal.**newThermalConductivityModel**(value='Null', parent=None)

Create a ThermalConductivityModel node. value is the model type in 'Constant-Prandtl', 'PowerLaw', 'SutherlandLaw'. If parent is not None, attach it to parent node.

#### Parameters

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ThermalConductivityModel node (pyTree):

```

# - newThermalConductivityModel (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newThermalConductivityModel(value='Null'); Internal.printTree(n)
#>> ['ThermalConductivityModel',array('Null',dtype='|S1'),[0 son],
↳ 'ThermalConductivityModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newThermalConductivityModel(value='SutherlandLaw', parent=t);
↳ Internal.printTree(t)

```

Converter.Internal.**newViscosityModel**(value='Null', parent=None)

Create a ViscosityModel node. value is the model type in 'Constant', 'PowerLaw', 'SutherlandLaw'. If parent is not None, attach it to parent node.

#### Parameters

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ViscosityModel node (pyTree):

```
# - newViscosityModel (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newViscosityModel(value='Null'); Internal.printTree(n)
#>> ['ViscosityModel',array('Null',dtype='|S1'),[0 son],'ViscosityModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newViscosityModel(value='SutherlandLaw', parent=t)
```

---

Converter.Internal.**newTurbulenceClosure**(value='Null', parent=None)

Create a TurbulenceClosure node. value is the closure type in 'EddyViscosity', 'ReynoldStress', 'ReynoldsStressAlgebraic'. If parent is not None, attach it to parent node.

**Parameters**

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new TurbulenceClosure node (pyTree):

```
# - newTurbulenceClosure (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newTurbulenceClosure(value='Null'); Internal.printTree(n)
#>> ['TurbulenceClosure',array('Null',dtype='|S1'),[0 son],'TurbulenceClosure_t
↪']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newTurbulenceClosure(value='ReynoldsStress', parent=t)
```



`Converter.Internal.newTurbulenceModel(value='Null', parent=None)`

Create a TurbulenceModel node. value is the model type in 'Algebraic\_BaldwinLomax', 'Algebraic\_CebeciSmith', 'HalfEquation\_JohnsonKing', 'OneEquation\_BaldwinBarth', 'OneEquation\_SpalartAllmaras', 'TwoEquation\_JonesLauder', 'TwoEquation\_MenterSST', 'TwoEquation\_Wilcox'. If parent is not None, attach it to parent node.

#### Parameters

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new TurbulenceModel node (pyTree):

```
# - newTurbulenceModel (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newTurbulenceModel(value='TwoEquation_MenterSST'); Internal.
↪ printTree(n)
#>> ['TurbulenceModel', array('TwoEquation_MenterSST', dtype='|S1'), [0 son],
↪ 'TurbulenceModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newTurbulenceModel(value='OneEquation_SpalartAllmaras', parent=t)
```

`Converter.Internal.newThermalRelaxationModel(value='Null', parent=None)`

Create a ThermalRelaxationModel node. value is the model type in 'Frozen', 'ThermalEquilib', 'ThermalNonequib'. If parent is not None, attach it to parent node.

#### Parameters

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ThermalRelaxationModel node (pyTree):

```
# - newThermalRelaxationModel (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newThermalRelaxationModel(value='Null'); Internal.printTree(n)
#>> ['ThermalRelaxationModel',array('Null',dtype='|S1'),[0 son],
↪'ThermalRelaxationModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newThermalRelaxationModel(value='ThermalNonequilib', parent=t)
```

---

Converter.Internal.**newChemicalKineticsModel**(value='Null', parent=None)

Create a ChemicalKineticsModel node. value is the model type in 'Frozen', 'ChemicalEquilibCurveFit', 'ChemicalEquilibMinimization', 'ChemicalNonequilib'. If parent is not None, attach it to parent node.

#### Parameters

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ChemicalKineticsModel node (pyTree):

```
# - newChemicalKineticsModel (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newChemicalKineticsModel(value='Null'); Internal.printTree(n)
#>> ['ChemicalKineticsModel',array('Null',dtype='|S1'),[0 son],
↪'ChemicalKineticsModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newChemicalKineticsModel(value='ChemicalNonequilib', parent=t)
```

---

Converter.Internal.**newEMElectricFieldModel**(value='Null', parent=None)

Create a EMElectricFieldModel node. value is the model type in 'Constant', 'Frozen', 'Interpolated', 'Voltage'. If parent is not None, attach it to parent node.

**Parameters**

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node**Return type** pyTree node*Example of use:*

- Create a new EMElectricFieldModel node (pyTree):

```
# - newEMElectricFieldModel (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newEMElectricFieldModel(value='Null'); Internal.printTree(n)
#>> ['EMElectricFieldModel',array('Null',dtype='|S1'),[0 son],
↪ 'EMElectricFieldModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
n = Internal.newEMElectricFieldModel(value='Voltage', parent=t)
```

Converter.Internal.**newEMMagneticFieldModel**(value='Null', parent=None)

Create a EMMagneticFieldModel node. value is the model type in 'Constant', 'Frozen', 'Interpolated'. If parent is not None, attach it to parent node.

**Parameters**

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node**Return type** pyTree node*Example of use:*

- Create a new EMMagneticFieldModel node (pyTree):

```
# - newEMMagneticFieldModel (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newEMMagneticFieldModel(value='Null'); Internal.printTree(n)
#>> ['EMMagneticFieldModel',array('Null',dtype='|S1'),[0 son],
↪ 'EMMagneticFieldModel_t']
```

```
# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
z = Internal.newEMMagneticFieldModel(value='Interpolated', parent=t)
```

Converter.Internal.**newEMConductivityModel**(value='Null', parent=None)

Create a EMConductivityModel node. value is the model type in 'Constant', 'Frozen', 'Equilibrium\_LinRessler', 'Chemistry\_LinRessler'. If parent is not None, attach it to parent node.

**Parameters**

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new EMConductivityModel node (pyTree):

```
# - newEMConductivityModel (pyTree) -
import Converter.Internal as Internal

# Create a zone node
n = Internal.newEMConductivityModel(value='Null'); Internal.printTree(n)
#>> ['EMConductivityModel',array('Null',dtype='|S1'),[0 son],
↪ 'EMConductivityModel_t']

# Create a node and attach it to parent
t = Internal.newFlowEquationSet()
z = Internal.newEMConductivityModel(value='Chemistry_LinRessler', parent=t)
```

Converter.Internal.**newBaseIterativeData**(name='BaseIterativeData', parent=None)

Create a BaseIterativeData node. If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new BaseIterativeData node (pyTree):

```
# - newBaseIterativeData (pyTree) -
import Converter.Internal as Internal

# Create a zone node
n = Internal.newBaseIterativeData(name='BaseIterativeData', nsteps=100, itype=
↳ 'IterationValues'); Internal.printTree(n)
#>> ['BaseIterativeData',array([[100]],dtype='int32'),[1 son],'BaseIterativeData_t'
↳ ']
#>> |['_IterationValues',array(shape=(100,),dtype='int32',order='F'),[0 son],
↳ 'DataArray_t']

# Create a node and attach it to parent
t = Internal.newCGNSTree()
b = Internal.newCGNSBase('Base', 3, 3, parent=t)
n = Internal.newBaseIterativeData(name='BaseIterativeData', nsteps=100, itype=
↳ 'IterationValues', parent=b)
```

Converter.Internal.newZoneIterativeData(name='ZoneIterativeData', parent=None)

Create a ZoneIterativeData node. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ZoneIterativeData node (pyTree):

```
# - newZoneIterativeData (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newZoneIterativeData(name='ZoneIterativeData'); Internal.
↳ printTree(n)
#>> ['ZoneIterativeData',None,[0 son],'ZoneIterativeData_t']

# Attach it to a parent node
z = Internal.newZone('Zone', zsize=[[10],[2],[0]], ztype='Structured')
Internal.newZoneIterativeData(name='ZoneIterativeData', parent=z)
```

`Converter.Internal.newRigidGridMotion(name='Motion', origin=[0.,0.,0.], mtype='Null', parent=None)`

Create a RigidGridMotion node. `mtype` is the motion type ('ConstantRate', 'VariableRate'). If `parent` is not `None`, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **origin** (list of 3 floats) – coordinates of the origin point of the motion
- **mtype** (string) – motion type
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new RigidGridMotion node (pyTree):

```
# - newRigidGridMotion (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newRigidGridMotion(name='Motion', origin=[0.,0.,0.], mtype=
↳ 'ConstantRate'); Internal.printTree(n)
#>> ['Motion',None,[2 sons],'RigidGridMotion_t']
#>>   |['_OriginLocation',array(shape=(3,),dtype='float64',order='F'),[0 son],
↳ 'DataArray_t']
#>>   |['_RigidGridMotionType',array('ConstantRate',dtype='|S1'),[0 son],
↳ 'RigidGridMotionType_t']

# Attach it to a parent node
z = Internal.newZone('Zone', zsize=[[10],[2],[0]], ztype='Structured')
Internal.newRigidGridMotion(name='Motion', origin=[0.,0.,0.], mtype=
↳ 'ConstantRate', parent=z)
```

---

`Converter.Internal.newRigidGridMotionType(value='ConstantRate', parent=None)`

Create a RigidGridMotionType node. `value` is the motion type ('ConstantRate', 'VariableRate'). If `parent` is not `None`, attach it to parent node.

#### Parameters

- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new RigidGridMotionType node (pyTree):

```
# - newRigidGridMotionType (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newRigidGridMotionType(value='ConstantRate'); Internal.printTree(n)
#>> ['RigidGridMotionType',array('ConstantRate',dtype='|S1'),[0 son],
↳'RigidGridMotionType_t']

# Attach it to a parent node
z = Internal.newRigidGridMotion(name='Motion', origin=[0.,0.,0.], mtype='Null')
Internal.newRigidGridMotionType(value='ConstantRate', parent=z)
```

`Converter.Internal.newReferenceState(name='ReferenceState', parent=None)`  
 Create a ReferenceState node. If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ReferenceState node (pyTree):

```
# - newReferenceState (pyTree) -
import Converter.Internal as Internal

# Create a ReferenceState node
n = Internal.newReferenceState(name='ReferenceState'); Internal.printTree(n)
#>> ['ReferenceState',None,[0 son],'ReferenceState_t']

# Create a ReferenceState node and attach it to base
t = Internal.newCGNSTree()
b = Internal.newCGNSBase('Base', 3, 3, parent=t)
n = Internal.newReferenceState(name='ReferenceState', parent=b); Internal.
↳printTree(n)
```

`Converter.Internal.newConvergenceHistory(name='GlobalConvergenceHistory', value=0, parent=None)`

Create a ConvergenceHistory node. value is an iteration number. If parent is not None, attach it to parent node.

### Parameters

- **name** (string) – name of the node
- **value** (integer) – value of the node (iteration number)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ConvergenceHistory node (pyTree):

```
# - newConvergenceHistory (pyTree) -
import Converter.Internal as Internal

# Create a ConvergenceHistory node
n = Internal.newConvergenceHistory(name='ZoneConvergenceHistory', value=100);
↳Internal.printTree(n)
#>> ['ZoneConvergenceHistory',array([100],dtype='int32'),[0 son],
↳'ConvergenceHistory_t']

# Create a ConvergenceHistory node and attach it to base
t = Internal.newCGNSTree()
b = Internal.newCGNSBase('Base', 3, 3, parent=t)
n = Internal.newConvergenceHistory(name='GlobalConvergenceHistory', value=100,
↳parent=b)
```

`Converter.Internal.newFamily(name='Family', parent=None)`

Create a zone Family node. If parent is not None, attach it to parent node.

### Parameters

- **name** (string) – name of the family
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new zone Family node (pyTree):



```
# - newFamily (pyTree) -
import Converter.Internal as Internal

# Create a Family node
n = Internal.newFamily(name='Wing'); Internal.printTree(n)
#>> ['Wing',None,[0 son],'Family_t']

# Create a Family node and attach it to base
t = Internal.newCGNSTree()
b = Internal.newCGNSBase('Base', 3, 3, parent=t)
n = Internal.newFamily(name='Wing', parent=b)
```

Converter.Internal.**newFamilyBC**(*value='UserDefined', parent=None*)

Create a FamilyBC node. *value* is a BC type string. If *parent* is not None, attach it to parent node.

#### Parameters

- **value** (string) – BC type ('BCWall','UserDefined'...)
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new FamilyBC node (pyTree):

```
# - newFamilyBC (pyTree) -
import Converter.Internal as Internal

# Create a FamilyBC node
n = Internal.newFamilyBC(value='BCWall'); Internal.printTree(n)
#>> ['FamilyBC',array('BCWall',dtype='|S1'),[0 son],'FamilyBC_t']

# Create a FamilyBC node and attach it to Family
b = Internal.newFamily(name='FamInjection')
n = Internal.newFamilyBC(value='UserDefined', parent=b)
```

Converter.Internal.**newGeometryReference**(*value='Null', file='MyCAD.iges', parent=None*)

Create a GeometryReference node. *value* is a type of CAD ('NASA-IGES', 'SDRC', 'Unigraphics', 'ProEngineer', 'ICEM-CFD'). If *parent* is not None, attach it to parent node.

#### Parameters

- **value** (string) – CAD type
- **file** (string) – name of CAD file
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new GeometryReference node (pyTree):

```
# - newGeometryReference (pyTree) -
import Converter.Internal as Internal

# Create a GeometryReference node
n = Internal.newGeometryReference(value='ICEM-CFD', file='MyCAD.tin'); Internal.
↪printTree(n)
#>> ['GeometryReference',None,[2 sons],'GeometryReference_t']
#>>   |['_GeometryFormat',array('ICEM-CFD',dtype='|S1'),[0 son],
↪   |'_GeometryFormat_t']
#>>   |['_GeometryFile',array('MyCAD.tin',dtype='|S1'),[0 son],'GeometryFile_t
↪   |']

# Create a GeometryReference node and attach it to a family
b = Internal.newFamily(name='FamWall')
n = Internal.newGeometryReference(value='NASA-IGES', file='MyCAD.iges',↪
↪parent=b)
```

---

`Converter.Internal.newArbitraryGridMotion(name='Motion', value='Null', parent=None)`

Create a ArbitraryGridMotion node. value is the type of motion ('NonDeforming-Grid', 'DeformingGrid'). If parent is not None, attach it to parent node.

**Parameters**

- **name** (string) – name of the node
- **value** (string) – type of motion
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new ArbitraryGridMotion node (pyTree):

```
# - newArbitraryGridMotion (pyTree) -
import Converter.Internal as Internal

# Create a node
n = Internal.newArbitraryGridMotion(name='Motion', value='DeformingGrid');
↳Internal.printTree(n)
#>> ['Motion',None,[1 son],'ArbitraryGridMotion_t']
#>>   |['_['ArbitraryGridMotion',array('DeformingGrid',dtype='|S1'),[0 son],
↳'ArbitraryGridMotionType_t']]

# Attach it to a parent node
z = Internal.newZone('Zone', zsize=[[10],[2],[0]], ztype='Structured')
Internal.newArbitraryGridMotion(name='Motion', value='NonDeformingGrid',
↳parent=z)
```

`Converter.Internal.newUserDefinedData(name='UserDefined', value=None, parent=None)`

Create a UserDefinedData node to store user specific data. If parent is not None, attach it to parent node.

#### Parameters

- **name** (string) – name of the node
- **value** (string) – value of the node
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new UserDefinedData node (pyTree):

```
# - newUserDefinedData (pyTree) -
import Converter.Internal as Internal

# Create a UserDefinedData node
n = Internal.newUserDefinedData(name='UserDefined', value=None); Internal.
↳printTree(n)
#>> ['UserDefined',None,[0 son],'UserDefinedData_t']

# Create a UserDefinedData node and attach it to a Family node (just an example)
f = Internal.newFamily(name='FamInjection')
n = Internal.newFamilyBC(value='BCInflow', parent=f)
n = Internal.newUserDefinedData(name='.Solver#BC', value=None, parent=n)
```

`Converter.Internal.newGravity(value=[0.,0.,9.81], parent=None)`

Create a Gravity node. `value` is the gravity vector. If `parent` is not `None`, attach it to parent node.

### Parameters

- **value** (list of 3 floats) – gravity vector
- **parent** (pyTree node) – optional parent node

**Returns** created node

**Return type** pyTree node

*Example of use:*

- Create a new Gravity node (pyTree):

```
# - newGravity (pyTree) -
import Converter.Internal as Internal

# Create a gravity node
n = Internal.newGravity(value=[0.,0.,9.81]); Internal.printTree(n)
#>> ['Gravity',None,[1 son],'Gravity_t']
#>> | _['GravityVector',array(shape=(3,),dtype='float64',order='F'),[0 son],
↪ 'DataArray_t']

# Create a Gravity node and attach it to base
t = Internal.newCGNSTree()
b = Internal.newCGNSBase('Base', 3, 3, parent=t)
n = Internal.newGravity(value=[0.,0.,9.81], parent=b)
```

---

CHAPTER  
**FOUR**

---

**INDEX**

- genindex
- modindex
- search