## 4.3.4. Interacting with META

### 4.3.4.1. General

The basic idea behind the scripting language is to automate many repetitive and tedious procedures with the minimum user interaction and to perform specific tasks that are not covered by the standard META commands and META session capabilities. Some of the tasks that can be performed within scripting language are the following:

- Extracting any type of information from an already loaded model (data from nodes, elements, boundary elements, materials, models, coordinate systems and groups).

- Access and manipulate data from META entities like annotations, groups, cut planes and isofunctions.

- Manipulating 2D plots: create and select curves, access data from curves and their points and modify them.

- Creating new results (deformation-scalar-vector) on nodes and elements with custom calculated data.

- Creating and manipulate META 3D and 2D windows.

- Creating user-defined windows.

- Manipulating files and directories.

- Reading and writing ASCII and binary files.

- Creating and handling data in XML form.

- Use of  session commands.

- Use of system commands.

- Running automatically (after launching META) a series of functions.

- Communication with the interface (File Manager functions, Pick functions).

All these tasks that can interact with the model and META entities data are controlled through a series of META specific functions.

In order to use a META specific function in Python, the relevant module has to be imported from META. The names of the modules are the available categories in the drop-down menu in tab *Library* in *Functions List* window. The functions are then called as methods of the imported module.

In Python the meta structs are objects and their attributes can be used directly without having to define the variable type first.

For example to print the ids and names of the visible parts

```
# PYTHON script
import meta
from meta import parts

def main():
	pids = parts.VisibleParts()
	for pid in pids:
		print('Id: '+str(pid.id)+' Name: '+pid.name)
```

### 4.3.4.2. Handling data

#### 4.3.4.2.1. Introduction

Using the available META modules it is possible to access and manipulate the data of the various METAentities. Contrary to BETA Scripting, the attributes of the META objects can be accessed directly without the need to define the object type first (as it is needed to define the META struct first).

#### 4.3.4.2.2. META classes

The available META classes are the following.

The description of each class ant the attributes of the class objects can be found in Function List of Script Editor under the respective category relevant to each class

| Class | Category |
|---|---|
| Annotation | annotations |
| AnnotationGroup | annotations |
| Boundary | boundaries |
| CentroidScalar | results |
| CentroidVector | results |
| Color | windows |
| Connection | connections |
| CoordSystem | coordsystems |
| CornerScalar | results |
| Curve | plot2d |
| CurveGroup | plot2d |
| Deformation | results |
| Elem | elements |
| ElemCoordSystem | coordsystems |
| Group | groups |
| Image | visuals |
| Isofunction | isofunctions |
| Material | materials |
| Model | models |
| NodalScalar | results |
| NodalVector | results |
| Node | nodes |
| OverlayRun | overlay |
| Page | pages |
| Part | parts |
| PartLayer | parts |
| Plane | planes |
| Plot | plot2d |
| PlotAxis | plot2d |
| Point | plot2d |
| Resultset | results |
| SpreadsheetCell | spreadsheet |
| Video | visuals |
| Window | windows |

#### 4.3.4.2.3. Subclassing META classes

META classes can be subclassed as shown in the example in the description of each class in Function List of Script Editor. An example is given below:

```
# PYTHON script
import meta
from meta import *

class MyModel(models.Model):
	def __init__(self, r):
		super(MyModel,self).__init__(
			r.id,
			r.name,
			r.label,
			r.deck,
			r.active)
		self.function_labels = models.FunctionLabelsOfModel(r.id)

def main():
	model_id = 0
	r = models.ModelById(model_id)
	if r:
		print(r.id, r.name, r.label, r.deck, r.active)
		my_r = MyModel(r)
		print(my_r.id, my_r.name, my_r.label, my_r.deck,
my_r.active, my_r.function_labels)

if __name__ == '__main__':
	main()
```

### 4.3.4.3. Collect entities

#### 4.3.4.3.1. Introduction

In META there is a separate group of script functions for META entities. For example, there are Element functions to collect the elements, Node functions to collect the nodes, etc.

Additionally, there are the following generic functions **CollectEntities** to collect all the specified entities that are loaded in META:

| Function | Description |
|---|---|
| `CollectEntities` | Collect all existing entities of the specified type in the current META session |
| `CollectModelEntities` | Collect all existing entities of the specified type in a model |
| `CollectWindowEntities` | Collect all existing entities of the specified type in a window |

The types of entities that can be collected using these functions are:

ANNOTATIONs

BOUNDARIEs

COORDINATE SYSTEMs

CUT PLANEs

ELEMENTs (SHELLs, SOLIDs, BEAMs, etc)

GROUPs

ISOFUNCTIONs
MATERIALs
META WINDOW entities (WINDOWs, FRINGEs, COLORs)
MODELs
NODEs
PAGEs
PARTs
PLOT2d entites (PLOTs, CURVEs and POINTs)
RESULTSETs
FRINGEs
VISUAL entities (FRAMEs, IMAGEs, VIDEOs)

#### 4.3.4.3.2. Collect entities of the database

For collecting all entities, the appropriate functions are the functions with just the type of the entities to be collected, e.g. Annotations(), Boundaries(), LoadsOfBoundary(), Elements(), Nodes(), Parts(), Models(), Groups(),Resultsets(), Connections(),  CyclesList() etc. Please note that Groups refer to the entities Parts/Groups of Part Manager, Includes, Boundaries, Connections, Sets

In the case of model entities, the model id is given as argument.

To access the entities directly according to their id, the list with the model objects can be converted to a dictionary with the XxxListToDict() functions where the key is the entities' id, e.g. AnnotationsListToDict(), ElementsListToDict(), etc.

```
# PYTHON script
import meta
from meta import parts

def main():
	model_id = 0
	all_parts = parts.Parts(model_id)
	for p in all_parts:
		print(p.id)
		print(p.name)

	dict_parts = parts.PartsListToDict(all_parts)
	for key, p in dict_parts.items():
		print(key)
		print(p)

	p = dict_parts[ 10 ] #Get directly part with id 10
	print(p.name)


if(__name__ == '__main__'):
	main()
```

To get the entities types of a model, the functions XxxTypeXxx() are available, e.g. ElementsTypes(), CurvesTypesNastran(), DeformationTypes(), ScalarTypes(), VectorTypes() etc.

```
# PYTHON script
import meta
from meta import elements

def main():
	model_id = 0
	all_types = elements.ElementsTypes(model_id)
	for element_type in all_types:
		print(element_type)

if(__name__ == '__main__'):
	main()
```

To get the available results in a database the functions XxxTypes(), XxxTypesAll(), XxxTypesList() are available, e.g. DeformationTypes(), DeformationTypesAll(), DeformationTypesList(), ScalarTypesList(), VectorTypesList(), etc. To retrieve deformation results types of a database the following can be used.

```
# PYTHON script
import meta
from meta import results

def main():
	filename = 'X:/project.metadb'
	deck = 'METADB'
	all_types = results.DeformationTypes(filename, deck)
	for one_type in all_types:
		total = len(one_type)
		deform_type = one_type[0] # Deformation type
		print(deform_type)
		for i in range(1,total):
			state = one_type[i]
			print(state)
			print(i) # Id of the state


if(__name__ == '__main__'):
	main()
```

For curves types of Dyna, Pamcrash and Radios files the functions CurvesTypesDynaWithNames(), CurvesTypesPamcrashWithNames() and CurvesTypesRadiosWithNames() are available to get also their name.

#### 4.3.4.3.3. Collect entities according to their attributes (results, name, id, type, position, comments, failure, free, outer)

There are specific functions in order to collect entities according to one of their attributes, e.g. NodesByName(), NodesByField10(), ElementById(), CurvesByName(), ElementsByType(), FailedElements, FreeNodes(), OuterElements(), FollowNodes() etc.

```
# PYTHON script
import meta
from meta import parts
from meta import constants

def main():
	model_id = 0
	part_type =constants.PSHELL
	collected_parts = parts.PartsByType(model_id, part_type)
	for p in collected_parts:
		print(p.id,p.type,p.subtype,p.visible,p.name,p.mat_id,
p.shell_thick,p.model_id)

if(__name__ == '__main__'):
	main()
```

When the function returns only one entity (e.g. ElementById()), then it returns the meta object of this entity. When it may return more than one entities (e.g. ElementsByType()), then it always returns a matrix with meta objects(even when only one entity is returned).

For entities that need to be collected and which are neighbors of other entities, the functions NeighbourElements(),NeighbourMaterials(),NeighbourParts() etc can be used. Neighboring entities are those which are directly attached to the specified entity or calculated by the solver.

For entities that need to be collected according to their nearest relative position to one given point, the functions NearestNodeOfPart, NearestNodeOfGroup, NearestElementOfPart, NearestElementOfGroup, NearestNodeOfMaterial, NearestElementOfMaterial etc can be used.

When entities need to be collected according to results (deformation, scalar, vector) or according to more advanced filters, then the functions AdvFiltersOnXxx() and XxxFromAdvFilters() can be used, e.g. AdvFiltersOnNodes(), AdvFiltersOnPoints, ElementsFromAdvFilters(), etc.. The difference between AdvFiltersXxx() and XxxFromAdvFilters() functions is that in AdvFiltersXxx() the filters are given as a string argument to the functions, whereas in XxxFromAdvFilters() the execution of the script will stop and a window will open in order for the user to specify the advanced filters.

An easy way to build the string argument with the advanced filters for AdvFiltersXxx() is to apply an iFilter from GUI inside a META session. The string, that corresponds to the filters selected, will be shown as argument of the applied META command in the command line or in the current META_post.ses.

```
# PYTHON script
import meta
from meta import results
from meta import elements

def main():
	model_id = 0
	adv_filters = list()
	adv_filters.append('add:Parts:id:==1:Keep All')
	adv_filters.append('intersect:Elements:centroidfuncmax::Max 3')
	all_resultsets = results.Resultsets(model_id)
	result = all_resultsets[1]
	collected_elements = elements.AdvFiltersOnElements(model_id, 
adv_filters, result)
	for e in collected_elements:
		print(e.id)

if(__name__ == '__main__'):
	main()
```

```
# PYTHON script
import meta
from meta import plot2d

def main():
	window_name = 'Window1'
	collected_curves = plot2d.CurvesFromAdvFilters(window_name)
	for c in collected_curves:
		print(c.id,c.name)

if(__name__ == '__main__'):
	main()
```

#### 4.3.4.3.4. Collect entities from other entities

There are many functions to collect entities from other entities, e.g. ElementsOfPart(), NodesOfElement(), ElementsOfNode(), PartsOfGroup(), etc. There is a separate function for each combination of the entities to collect and the entity that these belong to.

There are also functions that collect only the entities with specific attributes from other entities, e.g. ElementsOfNodeByType(), VisiblePartsOfGroupByType(), SelectedCurvesOfWindow(), etc.

```
# PYTHON script
import meta
from meta import parts

def main():
	model_id = 0
	group_name = 'My_Group'
	group_parts = parts.PartsOfGroup(model_id, group_name)
	for p in group_parts:

	print(p.id,p.type,p.subtype,p.visible,p.name,p.mat_id,p.shell_thic
k,p.model_id)

if(__name__ == '__main__'):
	main()
```

#### 4.3.4.3.5. Collect visible entities

For collecting the visible entities, the appropriate functions are the functions which include the word visible, e.g. VisibleAnnotations(), VisibleElements(), VisibleCurves(), etc.

There are also functions that collect only the visible entities with specific attributes, or only the visible entities from another entity, or only the visible entities with specific attributes from another entity, e.g. VisibleElementsByType(), VisiblePartsOfGroup(), VisiblePartsOfGroupByType(), VisibleCurvesOfPlot(), etc.

```
# PYTHON script
import meta
from meta import parts

def main():
	model_id = 0
	group_name = 'My_Group'
	window_name = 'MetaPost'
	visible_parts = parts.VisiblePartsOfGroup(model_id, group_name,
window_name)
	for p in visible_parts:
		print(p.id,p.type,p.subtype,p.visible,p.name,p.mat_id,
p.shell_thick,p.model_id)


if(__name__ == '__main__'):
	main()
```

#### 4.3.4.3.6. Collect identified entities

For getting only the identified entities of a model the respective functions are IdentifiedXxx(), e.g. IdentifiedElements(), IdentifiedNodes(), IdentifiedParts(), etc.

#### 4.3.4.3.7. Collect models information

There is a series of functions to get information from the loaded models, like NumOfNodes(), NumOfModels(), NumOfPartsByType(), etc.

Also information only for the active Models, Pages, Plots, Plot axes and Windows can be retrieved through the functions ActiveModels(), ActivePages(), ActivePlots(),ActivePlotAxes() and ActiveWindows().

```
# PYTHON script
import meta
from meta import models

def main():
	act_models = models.ActiveModels()
	for r in act_models:
		print(r.id,r.name,r.label,r.deck,r.active)

if(__name__ == '__main__'):
	main()
```

Information about the currently loaded plot models (the plot files listed in the *Read Results > Curves > Files List*) that may exist can be retrieved through the function PlotModels().

```
# PYTHON script
import meta
from meta import plot2d

def main():
	plot_models = plot2d.PlotModels()
	for pmod in plot_models:
		print(pmod.id,pmod.deck,pmod.filename)

if(__name__ == '__main__'):
	main()
```

#### 4.3.4.3.8. Collect newly created entities

For collecting all newly created entities of a specific type, the respective functions to use are

*CreateNewXxxStart()*, *ReportNewXxx()* and *CreateNewXxxEnd()*

The syntax is, for example: CreateNewAnnotationsStart(), ReportNewAnnotations(), CreateNewAnnotationsEnd(), etc.

This can be very useful in order to get the number and ids of entities created through filters or session Commands.

For example, for curves created through the Curve Function *User Defined* the function ReportNewXxx() will return the entity's structs without ending the recording of the created entities.

To collect newly created entities within a period, the functions to use are CollectNewXxxStart() and

CollectNewXxxEnd()

```
# PYTHON script
import meta
from meta import utils
from meta import plot2d

def main():
	utils.MetaCommand('xyplot read lsdyna "Window1" "Z:/demo/nodout" Node 1 Magnitude_of_acceleration_(ma)')
	plot2d.CollectNewCurvesStart()
	utils.MetaCommand('xyplot curve function userdef "Displacement Difference" "c1.x" "c1.y-c2.y" "Window1"')
	new_curves =plot2d.CollectNewCurvesEnd()
	for c in new_curves:
		print(c.id,c.name)

if(__name__ == '__main__'):
	main()
```

#### 4.3.4.3.9. Collect files and directories

For collecting files (of specific formats) or directories, the commands of the python os module can be used. A simple example to get the files in a directory is the following:

```
# PYTHON script
import meta
import os

def main():
	path = 'Z:/demo'
	files = os.listdir(path)
	for f in files:
		print(f)

if(__name__ == '__main__'):
	main()
```

An example to get the installation directory is:

```
# PYTHON script
import meta
import os
from meta import constants

def main():
	print(os.path.realpath(constants.app_root_dir))

if __name__ == '__main__':
	main()#
```

#### 4.3.4.3.10. Collect view parameters

For collecting view parameters, the functions XxxOfView() can be used. An example is the following:

```
# PYTHON script
import meta
from meta import windows
```

```
from meta import utils

def main():
	window_name = 'MetaPost'
	view_name = 'view0'

	#Save view
	utils.MetaCommand('view save "'+view_name+'"')

	#Get view parameters
	camera_position = windows.CameraPositionOfView(window_name,
view_name)
	reference_position = windows.ReferencePositionOfView(window_name,
view_name)
	up_vector = windows.UpVectorOfView(window_name, view_name)
	front_clipping_distance =
windows.FrontClippingDistanceOfView(window_name, view_name)
	back_clipping_distance =
windows.BackClippingDistanceOfView(window_name, view_name)
	viewing_angle = windows.ViewingAngleOfView(window_name, view_name)
	perspective_mode = windows.PerspectiveModeOfView(window_name,
view_name)

	#Change view parameters
	camera_position[ 0 ] = camera_position[ 0 ] + 100.0

	#Apply view
	if camera_position and reference_position and up_vector and
front_clipping_distance and back_clipping_distance and viewing_angle:
		utils.MetaCommand('view set '+str(camera_position[ 0
])+','+str(camera_position[ 1 ])+','+str(camera_position[ 2
])+','+str(reference_position[ 0 ])+','+str(reference_position[ 1
])+','+str(reference_position[ 2 ])+','+str(up_vector[ 0
])+','+str(up_vector[ 1 ])+','+str(up_vector[ 2
])+','+str(reference_position[ 0 ])+','+str(reference_position[ 1
])+','+str(reference_position[ 2
])+','+str(front_clipping_distance)+','+str(back_clipping_distance)+','+s
tr(viewing_angle)+','+str(perspective_mode))

if __name__ == '__main__':
	main()
```

#### 4.3.4.3.11. Select files or directories through the file manager

Scripting language interacts directly with the File Manager through the commands ***SelectOpenDir***, ***SelectSaveDir, SelectOpenFile, SelectSaveFile, SelectOpenFileIn, SelectSaveFileIn***. These functions open the File Manager and allow the selection or creation of files and directories. This is an elegant way to use file and directory paths in user scripts, since it enables the interactive definition of script parameters. The functions that deal with files return a matrix containing strings that represent the full path to the selected files, while those for directories return a string indicating the full path to the folder.

```
# PYTHON script
import meta
from meta import utils

def main():
	print('Select the file for reading')
	read_file = utils.SelectOpenFile(0, 'csv files (*.csv)')
	if not read_file:
		print('No file was selected')
	else:
		print('The file that was selected is: '+read_file[0])
	print('Select the log file for writing the error messages')
	save_file = utils.SelectSaveFile()
	if not save_file:
		print('No file was selected')
	else:
		print('The file that was selected for writing errors is: 
'+save_file[0])
	print('Select the directory where the META files are located');
	dir = utils.SelectOpenDir('');
	if not dir:
		print('No dir was selected')
	else:
		print('The selected directory is: '+dir)

if(__name__ == '__main__'):
	main()
```

If nothing is selected, it can be identified using the if not statement.

#### 4.3.4.4. Create, Edit, Delete and Handle Entities

##### 4.3.4.4.1. Introduction

The entities Annotations, Cut planes, Groups, Isofunctions, Windows, Models, Pages, Plots, Curves, Images and Videos can be created edited and deleted through script functions. The entities nodes, elements, parts, materials and boundaries CANNOT be created, edited or deleted in META in general, so this is also not possible through scripitng. Coordinate systems can be created but cannot be deleted. Moreover, specific functions exist for other operations related to the entities, e.g. to show, hide, get results from them, etc.

##### 4.3.4.4.2. Create entities

For creating entities the respective functions are CreateXxx(), e.g. CreateEmptyAnnotation(), CreatePlane(), CreateGroupFromElements(), etc..

The entities can also be created by applying the respective session commands (see 4.3.3. META session commands within scripts). However, one advantage of creating the entities through script functions is that the function returns the structs of the created entities. For example, when creating a curve through scripting, the id of the curve is available.

```
# PYTHON script
import meta
from meta import annotations

def main():
	window_name = 'MetaPost'
```

```
        text = '10th Annotation'
        a = annotations.CreateEmptyAnnotation(window_name, text)
        if (annotations.IsValidAnnotation(a)):
                print(a.id);

if(__name__ == '__main__'):
        main()
```

#### 4.3.4.4.3. Edit entities

There is a series of script functions for editing directly entities and their settings, for example SetSettingsOfAllCurves(),SetSettingsOfAllAnnotations(), SetAnnotationPointerOnlement(), ChangeOriginOfPlane(), AddPartsOnGroup(), RotateView(), etc. However, the full possibilities of editing entities in META are covered through the META session commands and can be achieved by using the MetaCommand syntax (see 4.3.3. META session commands within scripts).

```
# PYTHON script
import meta
from meta import planes
from meta import utils

def main():
        plane_name = 'plane_axis2'
        #Change origin through script function
        xorig = 1.26
        yorig = 7.52
        zorig = 3.59
        planes.ChangeOriginOfPlane(plane_name, xorig, yorig, zorig)
        #Change section width through session command
        utils.MetaCommand('plane options width 3 "'+plane_name+'"')

if(__name__ == '__main__'):
        main()
```

#### 4.3.4.4.4. Delete entities

For deleting entities the respective functions are DeleteXxx(), e.g. DeleteAnnotation(), DeleteCurve(), DeleteModel(), etc.

The entities can also be deleted by applying the respective session commands (see 4.3.3. META session commands within scripts).

#### 4.3.4.4.5. Show / Hide entities

For showing / hiding entities the respective functions are ShowXxx() and HideXxx(), e.g. ShowAnnotation(), HideCurve(), ShowModel(), etc.

When more than one 3d model entities or curves need to be shown / hidden, it is much faster to use the functions ShowSomeXxx() and HideSomeXxx(), e.g. ShowSomeElements(), HideSomeParts(), ShowSomeCurves(), etc. In this case the entities must be given as matrix.

The entities can also be shown / hidden by applying the respective session commands (see 4.3.3. META session commands within scripts).

#### 4.3.4.4.6. Identify entities

For identifying 3d model entities the respective functions are IdentifyXxx(), e.g. IdentifyNode(), IdentifyElement(), IdentifyMaterial(), etc.

When more than one 3d model entities need to be identified, it is much faster to use the functions IdentifySomeXxx(), e.g. IdentifySomeElements(), IdentifySomeParts(), etc. In this case the entities must be given as matrix.

The entities can also be identified by applying the respective session commands (see 4.3.3. META session commands within scripts).

#### 4.3.4.4.7. Get/handle material properties

To get the properties of a material the script function PropertyOfMaterial() can be used. To set these attributes the script function SetPropertyOfMaterial() can be used. If “all” is used as argument for property type in the PropertyOfMaterial() function then all material information lines will be returned, as in the input deck.

```
# PYTHON script
import meta
from meta import materials

def main():
	model_id = 0
	material_id = 1
	property_type = 'all'
	property_value = materials.PropertyOfMaterial(model_id,
material_id, property_type)
	print(property_value)

if(__name__ == '__main__'):
	main()
```

#### 4.3.4.4.8. Get results/attributes of entities

First of all, to get the states for which the resultsets are needed, the functions that can be used are CurrentResultset(), ResultsetsXxx(), FilterResultsetsXxx(), GeneratedResultsetsXxx().

If more than one labels exist for the resultset, the functions DeformationLabelsOfResultsets(), FunctionLabelsOfResultsets() can be used to get directly all the label results of the state. Alternatively, the functions StringDeformationLabelsOfResultsets(), StringFunctionLabelsOfResultsets() can be used to get the available labels and then the functions GetResultsetFromDeformationLabel() and GetResultsetFromFunctionLabel() can be used to get the specific label result of the state.

After the needed resultset is found, the functions to get the loaded results depend on the entity from which the result which is needed. So, in the Nodes functions group there are functions to get results on the nodes, in the Elements functions group there are functions to get results on elements, in the Parts functions group functions to get all results from a part, in the Materials functions group functions to get all results from a material, in the Groups functions group functions to get all results from a group and in the Models functions group there are functions to get all results from a model.

Moreover, there is a different function for each specific result needed. For nodes, to get the deformation results, there is the function DeformationOfNode(), to get the coordinates CoordinatesOfNode(), to get scalar results NodalScalarOfNode() and to get vector results NodalVectorOfNode().

Similarly for results on elements, there are functions DeformationsOfElement(), MaxDeformationOfElement(), CentroidScalarOfElement(), CornerScalarOfElement(), NodalScalarsOfElement() and more.

For parts / materials / models there are functions that return the results on all the nodes or elements, e.g. CentroidVectorOfPart(), and functions that return directly the maximum and minimum result, e.g. MinMaxNodalScalarOfModel().

For CELAS spring connection elements the functions StiffnessOfElasElement, DampingOfDampElement and MassOfMassElement can be used to get information about the stiffness, damping and mass respectively.

To get and set the number of steps of an axis the functions StepsOfPlotAxis() and SetStepsOfPlotAxis() respectively can be used.

To get, calculate, and set the material tension, compression, shear, x_tension, y_tension, x_compression, y_compression, shear and f12 limits of composites materials, the functions MaterialLimitOfMaterial() and MaterialLimitOfPart() or AddMaterialLimitOfMaterial() and AddMaterialLimitOfPart() can be used respectively.

*Remarks*

When the function returns only one entity (e.g. NodalScalarOfNode()), then it returns the meta_struct of this entity, when it may return more than one entities (e.g. DeformationsOfElement()), then it always returns a matrix with meta_structs (even when only one entity is returned).

```
# PYTHON script
import meta
from meta import results
from meta import nodes

def main():
	model_id = 0
	result = results.CurrentResultset(model_id)
	node_id = 10
	deform = nodes.DeformationOfNode(result, node_id)
	if(results.IsValidDeformation(deform)):
		print(deform.x,deform.y,deform.z,deform.total,
deform.node_id)

if(__name__ == '__main__'):
	main()
```

```
# PYTHON script
import meta
from meta import results
from meta import models

def main():
	model_id = 0
	all_resultsets = results.Resultsets(model_id)
	result = all_resultsets[2]
	print(result)
	nodal = models.MinMaxNodalVectorOfModel(result)
	print(len(nodal))
	if(len(nodal)):
		min_nodal = nodal[0] #Struct with the minimum nodal vector
value
		print(min_nodal.value) #Minimum nodal vector value
		print(min_nodal.x,min_nodal.y,min_nodal.z) #Normalized
coordinates (X, Y, Z) of the minimun nodal vector value
		print(min_nodal.node_id) #Id of the node with the minimum
nodal vector value
```

```
            print(min_nodal.part_id); #Id of the part or -1 if no part
exists

            max_nodal = nodal[1] #Struct with the maximum nodal vector
value
            print(max_nodal.value) #Maximum nodal vector value
            print(max_nodal.x,max_nodal.y,max_nodal.z) #Normalized
coordinates (X, Y, Z) of the maximum nodal vector value
            print(max_nodal.node_id) #Id of the node with the maximum
nodal vector value
            print(max_nodal.part_id) #Id of the part or -1 if no part
exists


if(__name__ == '__main__'):
      main()
```

#### 4.3.4.4.9. Get measurements from entities

The functions to get measurements depend on the entity from which the measurement is needed. So, in the Nodes functions group there are functions to get measurements from the nodes, in the Elements functions group there are functions to get measurements from the elements, in the Parts functions group functions to get measurements from a part and in the Groups functions group functions to get measurements from a group.

Moreover, there is a different function for each specific measurement needed. To get the distance between a  node and another node there is the function DistanceNodeToNode(), to get the distance between a node and a part the function is DistanceNodeToPart(), etc. To get the angle formed by nodes the function is AngleOfNodes(). To get the distance between a part and a group the function is DistancePartToGroup().

```
# PYTHON script
import meta
from meta import results
from meta import parts

def main():
      part_model = 0
      all_resultsets =results.Resultsets(part_model)
      part_result = all_resultsets[1]
      part_type = 13
      part_id = 1
      group_model = 0
      group_result = all_resultsets[2]
      group_name = "My_Group"

      distance = parts.DistancePartToGroup(part_model, part_result,
part_type, part_id, group_model, group_result, group_name)
      if(len(distance)):
            dist_x = distance[0] # Distance in direction X
            dist_y = distance[1] # Distance in direction Y
            dist_z = distance[2] # Distance in direction Z
            dist_total = distance[3] # Total distance
            print(dist_x,dist_y,dist_z,dist_total)

if(__name__ == '__main__'):
      main()
```

#### 4.3.4.4.10. Get connected and neighbour elements/nodes

In cases that the connected elements or nodes are needed, the functions NeighbourElementsXxx(), NodesOfElements() and ElementsOfNodes() should be used. In case of mesh independent spotweld connections, where there is no node connectivity, the functions NeighbourElementsXxx() can still be used as long as the connectivity is supported in META.

The above script functions should be preferred from the visibility session commands, e.g. “add connected”, as the first are much faster.

```
# PYTHON script
import meta
from meta import elements

def main():
	model_id = 0
	element_type = 3
	element_id = 1
	second_id = -1
	neighbour_type =3
	neighbour_elements = elements.NeighbourElementsByType(model_id,
element_type, element_id, second_id, neighbour_type)
	for e in neighbour_elements:
		print(e.id)

if(__name__ == '__main__'):
	main()
```

In cases that the connected elements, nodes, parts etc to connections are needed, the functions ConnectedXxxOfConnection() should be used.

```
# PYTHON script
import meta
from meta import connections

def main():
	model_id = 0
	connection_id = 100001
	connected_elements =
connections.ConnectedElementsOfConnection(model_id, connection_id)
	for e in connected_elements:
		print(e.id)

if(__name__ == '__main__'):
	main()
```

#### 4.3.4.4.11. Get spreadsheet entities/data

There are functions to get data stored in a spreadsheet. To get a cells data the function SpreadsheetCellByRowColumn(sheet_name, row, col) can be used.

Also the entire row's or column's data can be stored in a matrix through the functions SpreadsheetCellsByRow(sheet_name, row). and SpreadsheetCellsByColumn(sheet_name, col). Returned matrix stops at the last non-empty cell. If all cells are empty, an empty matrix is returned)

To get cells data of an entire cells area the functions SpreadsheetCellsByArea(sheet_name, top_row, left_col, bottom_row, right_col) and SpreadsheetCellsByLabel(sheet_name, cell_label) can be used. Cells data are stored in a matrix of same size as the defined spreadsheet area.

Also in order to identify the spreadsheet cells area that involves data the function SpreadsheetBoundingArea(sheet_name) can be used. A matrix that determines the bounding box of non-empty cells (matrix[0][0]: top-row, matrix[0][1]: left-column, matrix[1][0]: bottom-row, matrix[1][1]: right-column) will be returned.

#### 4.3.4.4.12. Get attributes of user Toolbars

To get the attributes of specific entities of user Toolbars the functions CheckboxStateOfToolbar(), SliderValueOfToolbar() and TextboxValueOfToolbar() are available.

```
# PYTHON script
import meta
from meta import utils

def main():
	toolbar_name = 'Toolbar 1'
	checkbox_name = 'Checkbox 1'
	state = utils.CheckboxStateOfToolbar(toolbar_name, checkbox_name)
	if state == 1:
		print('Checked')
	elif state == 0:
		print('Not checked')
	elif state == -1:
		print('Failure!')


if(__name__ == '__main__'):
	main()
```

### 4.3.4.5. Load Geometry and Solver Results, Create User-Defined Results

#### 4.3.4.5.1. Load geometry and field / history results

For loading geometry the respective functions are LoadModel() for solver geometry data files and LoadProjectModel() for geometries from META databases and META projects.

For loading field results the functions are LoadDeformations(), LoadModalDeformations(), LoadScalar() and LoadVector() for solver results data files and LoadProjectDeformations(), LoadProjectScalar() and LoadProjectVector() for results from META databases and META projects.

The argument "data" which is needed for the functions is the same as the argument used in the equivalent META session commands. The best way to retrieve the correct syntax is to load the result manually from the META interface and view the session command passed to the current META_post.ses file.

In order to append the loaded geometry/results the functions LoadAppendDeformations(), LoadAppendProjectDeformations(), LoadAppendScalar(), LoadAppendScalarGetCoordSystems(), LoadAppendProjectScalar(), LoadAppendVector(), LoadAppendVectorGetCoordSystems(), LoadAppendProjectVector() are available.

For loading curves from history results files the respective functions are LoadCurvesXxx(), e.g. LoadCurvesNastran(), LoadCurvesDyna().

The arguments which are needed for the functions, e.g. entities, variables, etc, are the same as the arguments used in the META session commands. The best way to retrieve the correct syntax is to load the result manually from the META interface and view the session command passed to the current META_post.ses file. The only difference is that the results are passed as matrices to the script functions. The string expressions can be easily changed to matrices and vice versa through the functions RangeToMatrix() and MatrixToRange().

Especially for loading curves from RADIOSS history results files, the function LoadCurvesRadioss() needs the time history id as argument. In case the time history id is not known, strings 'all' or '*' can be used referring to all available time history ids or the function GetRadiossTimeHistoryId() can be used to find the time history id for a specific history result.

The entities can also be loaded by applying the respective session commands (see also chapter "META session commands for use in Scripting Language"). However, one advantage of loading the entities through script functions is that the function returns the structs of the created entities. For example, when loading a model through the script function, the id of the model is available.

```
# PYTHON script
import meta
from meta import models
from meta import results

def main():
	window_name = 'MetaPost'
	filename = '/home/demo/example.op2'
	deck = 'NASTRAN'
	r = models.LoadModel(window_name, filename, deck)
	if models.IsValidModel(r):
		model_id = r.id
		states = '1-3'
		data = 'Displacements,Translational'
		new_resultsets =results. LoadDeformations(model_id,
filename, deck, states, data)
		data = 'Stresses,VonMises,MaxofTopBottom'
		new_resultsets = results.LoadScalar(model_id, filename,
deck, states, data)
		for res in new_resultsets:

	print(res.name,res.nodal_data_name,res.function_data_name,res.stat
e)


if(__name__ == '__main__'):
	main()
# PYTHON script
import meta
from meta import plot2d

def main():
	window_name = 'Window1'
	plot_id = 0
	filename = '/home/demo/exmampleT01'
	entity_id = '12954'
	entity_type = 'node'
	variable = ""
	time_history = [ plot2d.GetRadiossTimeHistoryId(filename,
entity_id, entity_type, variable) ]
	#time_history = ['all' ]
	entities = [ entity_id ]
	variables = [ 'dx' ]
	new_curves = plot2d.LoadCurvesRadioss(window_name, plot_id,
filename, entity_type, time_history, entities, variables)
	for c in new_curves:
		print(c.id)

if(__name__ == '__main__'):
	main()
```

***Remarks***

The deck of the results is given as argument in the functions to load geometry and field results, whereas different functions exist for each deck for loading history results.

#### 4.3.4.5.2. Create user-defined field results

Apart from loading solver results, it is possible to create user defined field results using the results from calculations realized inside the script.

First of all a new empty state for field results can be created with the function CreateResultSet().

To add a new label to the resultset and set the results to the nodes or the elements, the functions to be used are AddDeformationOnAllNodes(), AddNodalScalarOnAllNodes() and AddNodalVectorOnAllNodes().

For more complicated procedures:

To reset all the values of a resultset and add values to the nodes or the elements, the functions to be used are StartAddingXxx() and EndAddingXxx(), e.g. StartAddingDeformations(), EndAddindDeformations().

To change the values of a resultset the functions to be used are StartChangingXxx() and EndAddingXxx(), e.g. StartChangingCentroidVector(), EndAddingCentroidVector().

To append a new label to a resultset and set values to the nodes or the elements for this label, the functions to be used are StartAppendingXxx() and EndAddingXxx(), e.g. StartAppendingCentroidScalar(), EndAddingCentroidScalar().

Between the StartXxx() and EndXxx() functions the values can be assigned to the nodes with the functions AddCentroidScalarOnElement(), AddCentroidScalarOnSomeElements(), AddCentroidVectorOnElement(), AddCentroidVectorOnSomeElements(), AddCornerScalarOnElement(), AddCornerScalarOnSomeElements(), AddDeformationOnNode() and AddDeformationOnSomeNodes(). When values are to be to more than one elements / nodes, functions XxxOnSomeElements() and XxxOnSomeNodes() should be preferred, as they are much faster.

Improtant Note: The EndAddingXxx() function must be called at the end, or else the values will not be assigned to the resultset.

### 4.3.4.6. Useful notes about META script functions

#### 4.3.4.6.1. Speed up the execution of scripts

When executing scripts in META some points should be taken into consideration in order to achieve minimum execution times.

- When the script includes functions that change the display of the META windows, each redraw will result to time consumption. To speed up the execution of the script, the redraws can be disabled, and enabled only at the end of the script, in order to display directly the final state.

This can be achieved through the lines

```
MetaCommand('options session controldraw disable')

MetaCommand('options session controldraw enable')
```

- When a function needs to be applied on more than one entities, it is always faster to use the function that applies directly on all the needed entities XxxOnSomeXxx, e.g. IdentifySomeNodes(), AddElementScalarOnSomeElements(), etc, instead of applying the single function for each entitty.

- When all the results from entitiy are needed, it is faster to use the function that gets all the results directly, e.g. DeformationsOfModel, CentroidScalarOfPart(), etc, instead of getting the result for each element / node separately.

- In cases the connected elements or nodes are needed the functions NeighbourElementsXxx(), NodesOfElements() and ElementsOfNodes() should be preferred from the visibility session commands, e.g. “add connected”, as the first are much faster.

### 4.3.5. User defined buttons and custom GUI

#### 4.3.5.1. General

META scripting language enables the creation of user defined buttons and fully customized graphical interfaces. The buttons that can be created are similar to the buttons of the GUI and are used to invoke user functions. For the management of specific tasks that must be controlled through a number of definitions and actions, it is very useful to create a custom GUI. There are two libraries of functions to create custom user interface. The first supports only the basic widgets that a GUI can hold, like checkbuttons, radio buttons, menu buttons, lists, tables, while the second library, BCGUI functions, is more extensive and except from the main widgets mentioned before it also supports hundreds of more sophisticated tools. These tools allow the creation of any complex interface that may also contain tab widgets, popup menus, spin boxes, group of buttons, tooltips and so on. The first library is similar for Python and BETA Scripting language and is described in the relevant paragraph of the chapter *BETA Scripting Language*. The second library is described in a separate chapter for both Python and BETA Scripting Lanuage.