Scripting in Mechanical Guide

ANSYS, Inc.
Southpointe
2600 ANSYS Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
http://www.ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 2020 R1
January 2020

Copyright and Trademark Information

Disclaimer Notice

U.S. Government Rights

Third-Party Software

Published in the U.S.A.

Table of Contents

Scripting Quick Start

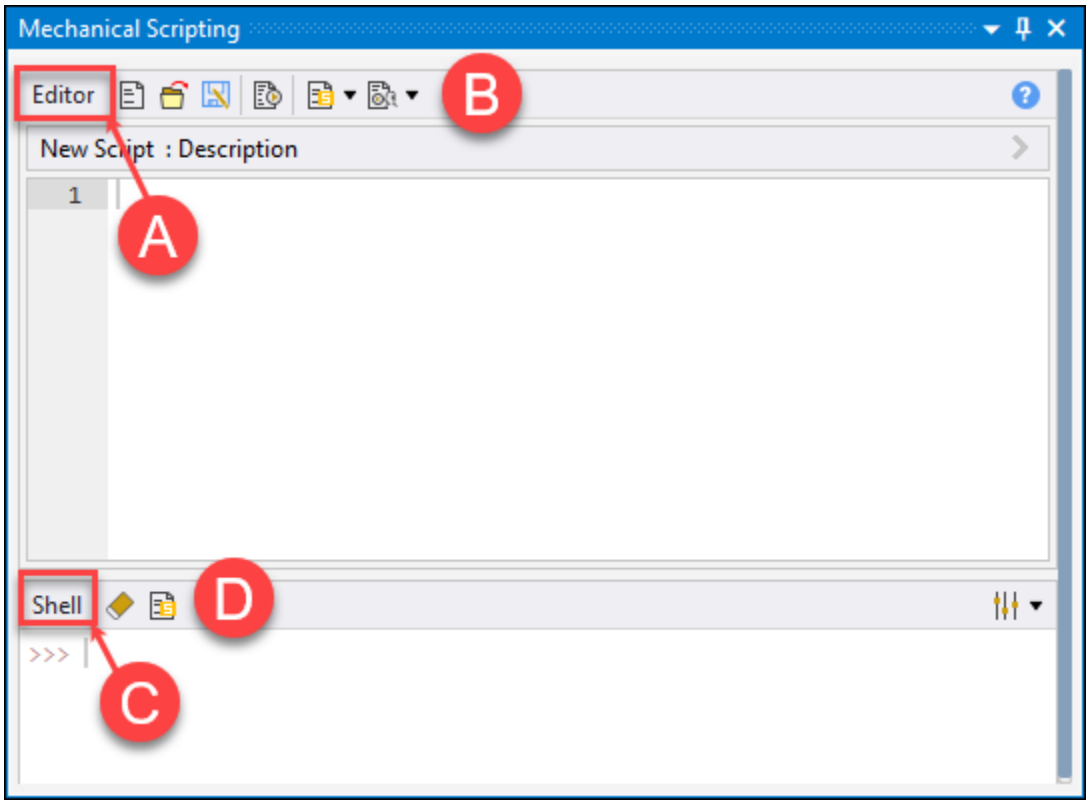
Scripting refers to the use of a programming language to interact with and modify a software product. Scripting can also be used to automate routine tasks. In ANSYS Mechanical, you can use ANSYS ACT and Mechanical Python APIs (Application Programming Interfaces).

Scripting Introduction

This section provides introductory information about scripting in Mechanical:

Mechanical Scripting View

The Mechanical **Scripting** view provides a multi-line editor and shell for APIs so that you can develop and debug scripts. You can show and hide the **Scripting** view from the ribbon's **Automation** tab. In the **Mechanical** group, clicking **Scripting** switches between showing and hiding this view.



Callout	Name	Description
A	Editor	Work on long scripts for your workflows.
B	Editor Toolbar	Interact with scripts by performing actions such as opening scripts from disk, promoting scripts to buttons, and more.

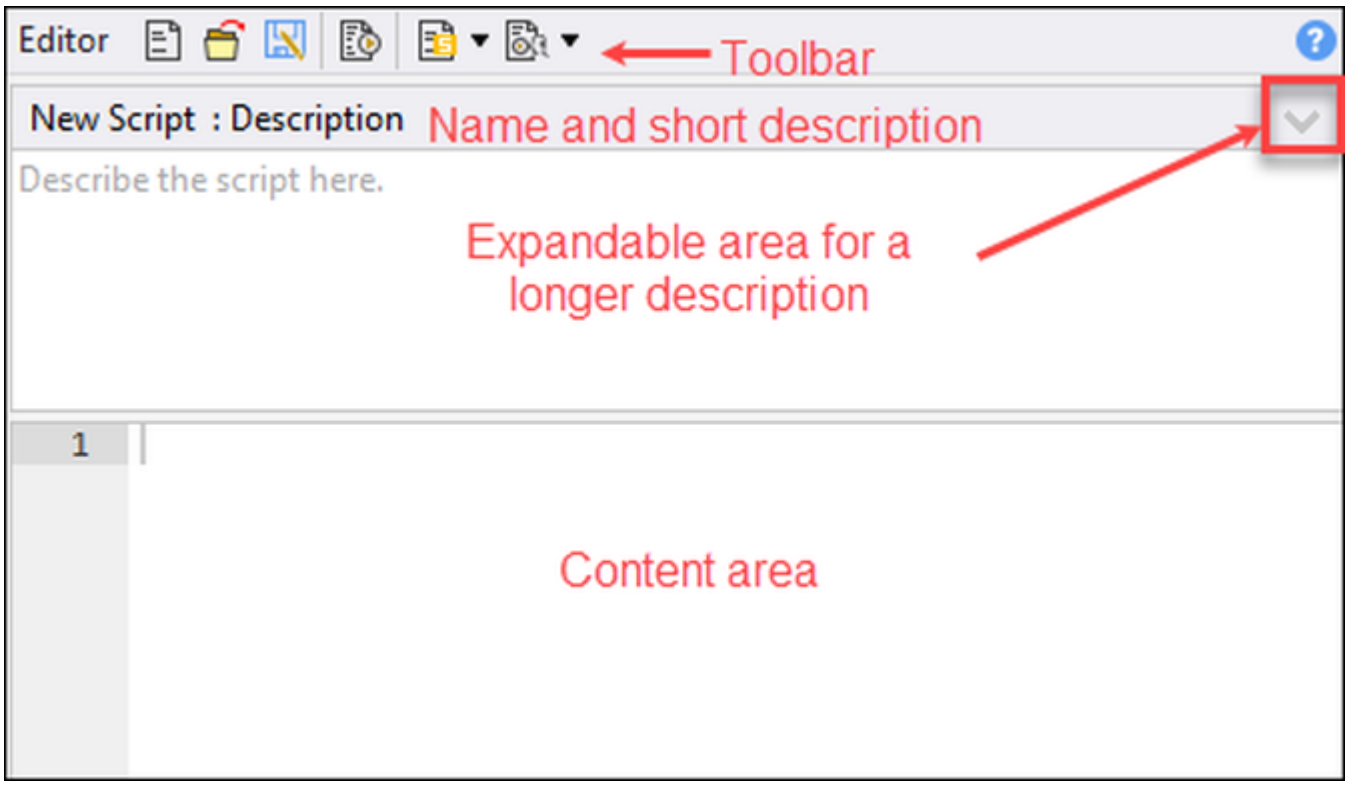
Callout	Name	Description
C	Shell	Work with shorter commands to build your scripts.
D	Shell Toolbar	Interface with the shell by performing actions such as clearing the shell and opening the snippet inserter.

Note:

- The **Scripting** view initially opens in a locked position to the right of the graphics view. You can click the header bar and drag this view to anywhere in Mechanical, dropping it into a new locked position. By double-clicking the header bar, you can toggle between the locked position and a floating window. You can resize and move the floating window as needed.
- If you have ACT extensions loaded and debug mode is enabled, below the **Shell** area, a tab displays for each loaded extension so that you can set the scope. For more information, see Scope Selection for ACT Extensions (p. 20).
- You can revert to the **ACT Console** by changing the scripting view preference under **File > Options > Mechanical > UI Options > New Scripting UI**. Mechanical must be restarted to see the scripting view change.

Working with the Editor

The **Editor** appears in the top panel of the Mechanical **Scripting** view.



Toolbar

The toolbar for the **Editor** provides several buttons:

1. **New Script**: Clears the name, description, and script boxes, providing you with a new script template in which to work.
2. **Open Script**: Opens a script, which is any Python file (*** . py**), from disk.
3. **Save Script As**: Saves the script that you are working on to disk.
4. **Run Script**: Executes the script.
5. **Snippet Inserter**: Opens the snippet inserter (p. 13).
6. **Promote Script to Snippet**: Opens the Snippet Editor (p. 14) with fields already filled out using the information from the script in the **Editor**.
7. **Open Snippet Editor**: Opens the **Snippet Editor** with empty fields so that you can create your own script.
8. **Button Editor**: Opens the **Button Editor**. For more information, see Creating User-Defined Buttons in the *Mechanical* User's Guide.
9. **Promote Script to Button**: Opens the **Button Editor** with fields already filled out using the information from the script in the **Editor**.
10. **Help**: Displays a list of keyboard shortcuts.

Name and Description Area

Below the toolbar is the area in which you enter a name and description for the script:

1. **Script name**: Initially, **New Script: Description** is shown. When you click **New Script**, the field becomes editable so that you can specify the name and a description for the new script. When you save or open a script, this field will display the file name.

2. Button for showing and hiding the area in which you describe the script.

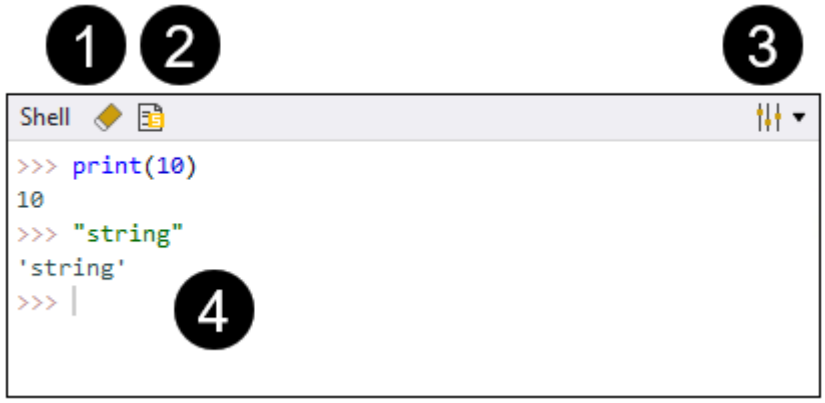
Content Area

The content area is where you write scripts. This area provides autocompletion (p. 8) and the following shortcuts:

- **Ctrl + f**: Opens the find and replace dialog box.
- **Ctrl + F5**: Executes the script.
- **Ctrl + i**: Opens the snippet inserter (p. 13).

Working with the Shell

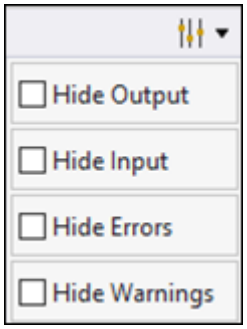
The **Shell** appears in the lower panel of the Mechanical **Scripting** view.



Toolbar

The toolbar for the **Shell** provides these buttons, which have these callotus in the previous figure:

1. **Clear Contents**: Clears the contents of the **Shell**. This can also be done by executing the command `clear` inside the **Shell**.
2. **Insert Snippet**: Opens the snippet inserter (p. 13) in context of the **Shell**.
3. **Shell Preferences**: Displays a drop-down menu for indicating whether to hide the output, input, errors, and warnings. All check boxes are cleared by default so that the output, input, errors, and warnings are all shown.



4. **Content area**: Area in which you enter and execute single-line or multi-line commands. This area provides autocompletion (p. 8).

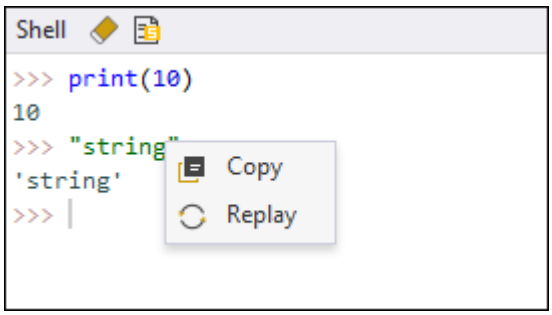
Color Coding for Output in the Shell

In the output, the color of the text distinguishes output from errors and warnings.

Text Color	Item
Gray	Output
Red	Error
Yellow	Warning

Context Menu

There is a context menu associated with the commands that have been executed. To display this context menu, place the mouse cursor over the executed command and right-click.



- **Copy**: Copies the command to the clipboard.
- **Replay**: Re-inserts the executed command in the content area for execution.

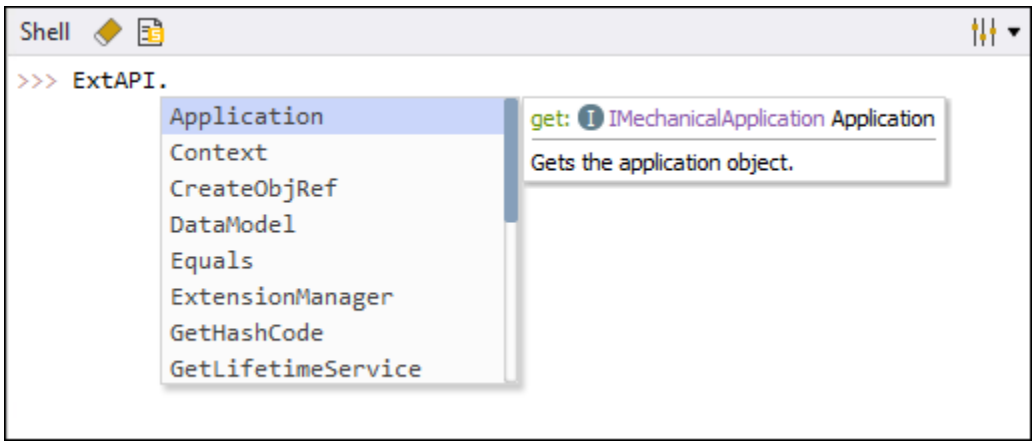
Shortcuts:

The **Shell** provides these shortcuts:

- **Ctrl** + ↑: Takes you to the previously executed command.
- **Ctrl** + ↓: Takes you to the next executed command.
- **Enter**: Executes the command.
- **Shift** + **Enter**: Inserts a new line.

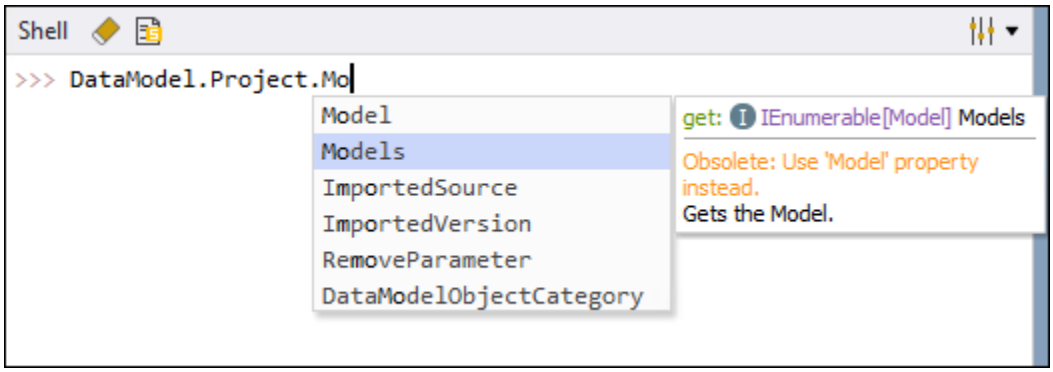
Autocompletion

The text editor in the **Editor**, Snippet Editor (p. 14), and the input field in the **Shell** all provide autocompletion.



Autocompletion Tooltips

When you place the mouse cursor over any property or method in the list of suggestions provided, the tooltip displays information about this item. The following image shows the tooltip for the property **`Models`**.



Tooltips use color-coding to indicate the syntax:

Color	Syntax
Green	Accessibility
Purple	Type
Orange	Warning
Blue	Argument

Properties

General formatting for properties follow:

- **get/set mode**: ***`ReturnType PropertyName`***
- Description of the property
- **Returns**: Description of what is returned (if any)
- **Remarks**: Additional information (if any)
- **Example**: Sample entry (if any)

Methods

General formatting for methods follow:

- ***`ReturnType MethodName`*** **(*`Argument Type Argument Name`*)**
- ***`Argument Name`***: Description of the argument
- **Returns**: Description of what is returned (if any)
- **Remarks**: Additional information (if any)
- **Example**: Sample entry (if any)

Additional Information

Tootltips can also provide:

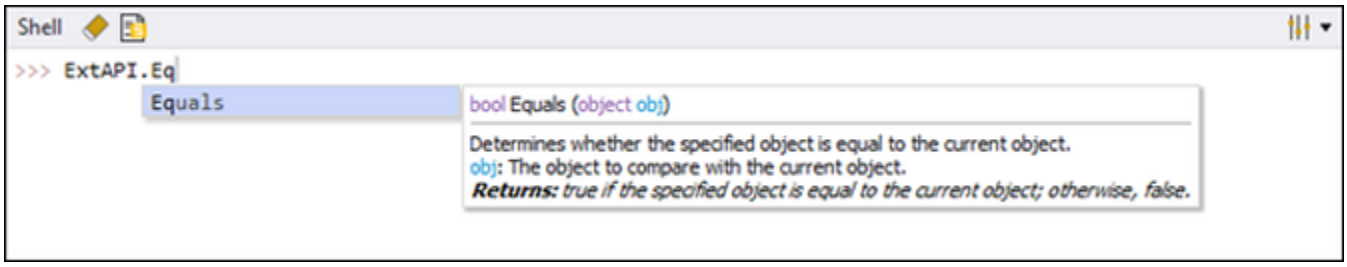
- **.NET** properties where applicable
- **Warning messages** when special attention is needed
- **Prototype information** for methods when cursor is inside brackets and indexers when cursor is inside square brackets
- Overloaded methods, including the following details:
  - Number of overloaded methods
  - Prototypes for overloaded methods (accessed by pressing the arrows in the prototype tooltip)

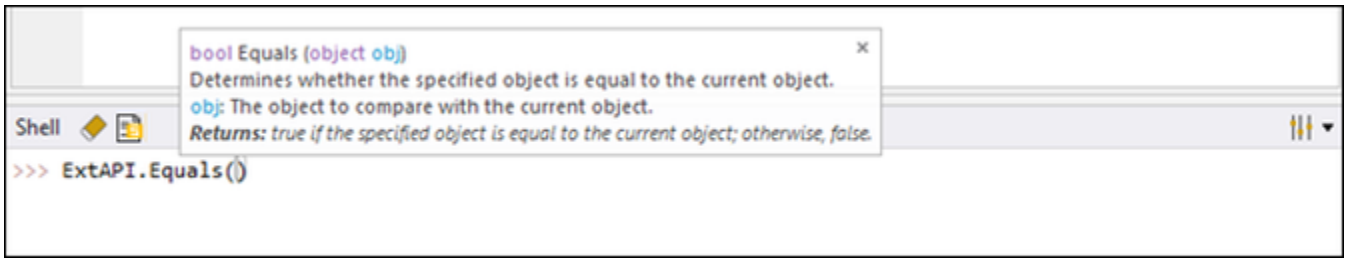
– Members for overloaded methods

  The tooltip for an overloaded method is a list of all members from all overloaded methods.

Tooltip Examples

The following images show two different tooltip examples for the method **`Equals`** in two different stages of using autocompletion:





Keyboard Shortcuts:

The tooltip provides these keyboard shortcuts:

- **Enter**: Inserts the suggestion.
- **Esc**: Closes autocompletion tooltip.
- **Ctl + Space**: Forces autocompletion to reopen.

Snippets

Snippets are code or code templates that you can quickly and easily insert in the command line, saving you from repetitive typing. As you write scripts, you can insert any of the snippets that ANSYS supplies. Additionally, you can begin building your own library of snippets to either supplement or replace supplied snippets with your own custom snippets.

For more information, see:

Supplied Snippets

Snippet Usage Example

Snippet Inserter

Snippet Creation and Management

Supplied Snippets

Descriptions follow of all supplied snippets:

The snippet **ExtAPI** inserts **`ExtAPI.`** in the command line, providing you with immediate access to the ACT API. From the autocomplete options in the tooltip, you can then begin selecting members to build your command.

The snippet **Snippet Template** provides sample code for swapping two variables. The comments explain how a snippet can contain editable fields, which are automatically selected for modification when the snippet is inserted. When a field is selected, you can type a new value to override the default value or let the default value remain. Pressing the **Tab** key moves the cursor to the next editable field.

The folder **Helpers** provides snippets for frequently used commands:

- **Project**. Inserts **`project = DataModel.Project`**, providing access to the project, which is the top level of the hierarchy in the Mechanical tree. To interact with first-level objects in the tree, you would then type a period and use the list of suggestion to select the attribute **`Model`** and then the attribute for the first-level object. For example, these command line entries provide access to the first-level tree objects **`Connections`** and **`Named Selections`**:

  – **`project=DataModel.Project.Model.Connections`**

  – **`project=DataModel.Project.Model.NamedSelections`**

  The snippets **Mesh** and **Geometry** provide examples of easier methods for accessing first-level objects.

- **Mesh**. Inserts **`mesh = Model.Mesh`**, providing access to the object **`Mesh`**.

- **Geometry**. Inserts **`geometry = Model.Geometry`**, providing access to the object **`Geometry`**.

- **Analysis**. Inserts **`analysis = Model.Analyses[0]`**, providing access to the first analysis of the model. As indicated earlier, Python starts counting at 0 rather than 1.

- **ObjectsByName**. Inserts **`solution = DataModel.GetObjectsByName("Solution")`**, providing access to a list of all solutions. To apply this function with pressure, you use **`solution = DataModel.GetObjectsByName("pressure")`**.

- **PathToFirstActiveObject**. Inserts **`path_to_object = Tree.GetPathToFirstActiveObject()`**, providing access to the full path that must be typed to go to the first object that is selected in the tree.

- **ObjectsByType**. Inserts **`coordinate_system_list = DataModel.GetObjectsByType(DataModelObjectCategory.CoordinateSystem)`**, where **`DataModelObjectCategory`** is the enumeration containing all types. To apply this function with pressure, you use **`pressure_list = DataModel.GetObjectsByType(DataModelObjectCategory.Pressure)`**.

- **Active Objects**. Inserts **`active_objects = Tree.ActiveObjects`**, providing a list of all selected objects in the tree.

- **FirstActiveObject**. Inserts **`first_active_object = Tree.FirstActiveObject`**, providing access to the first of all selected objects.

- **Quantity**. Inserts **`Quantity("1 [mm]"`**, providing a command method in which you can declare values. Three examples follow:

```
pres = DataModel.Project.Model.Analyses[0].AddPressure()
```

```
pres.Magnitude.Inputs[0].DiscreteValues = [Quantity('0 [sec]'), Quantity('1 [sec]'), Quantity('2 [sec]')]
```

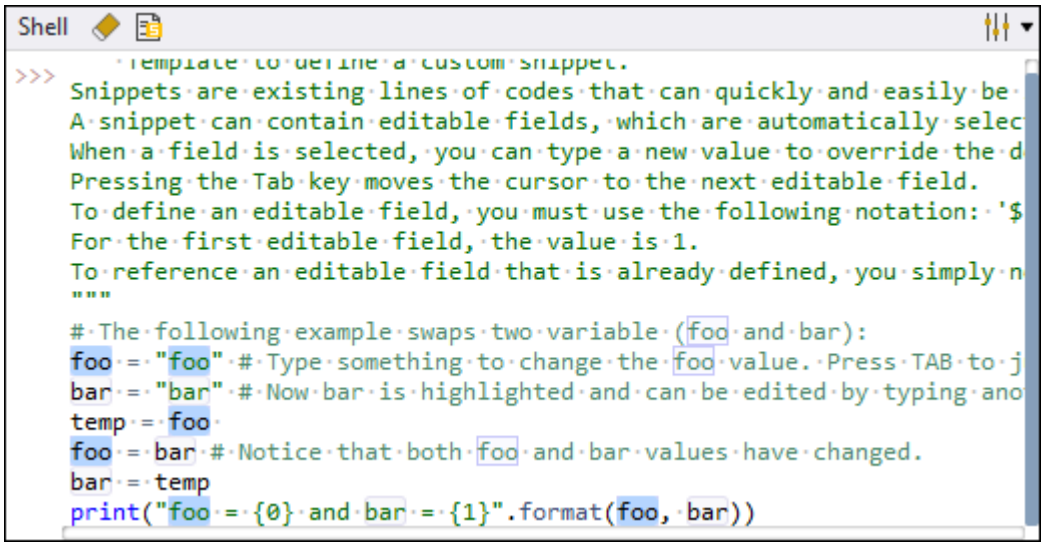
```
pres.Magnitude.Output.DiscreteValues = [Quantity('0 [Pa]'), Quantity('50 [Pa]'), Quantity('100 [Pa]')]
```

- **Selection Manager**. Inserts **`selection_manager = ExtAPI.SelectionManager`**, providing access to the Selection Manager. This object contains properties and functions related to graphical selection within Mechanical, such as getting information about the current selection or modifying the selection.
- **Model View Manager**. Inserts **`model_view_manager Graphics.ModelViewManager`**, providing access to the Model View Manager. This object provides functionality to control managed graphical views.

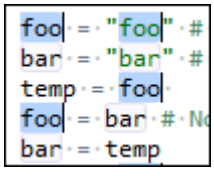
Lastly, the folder **Examples** provides snippets that you can use as templates. For example, the snippet **Add Pressure** shows how to create a pressure on the first face of the first body of the first part.

Snippet Usage Example

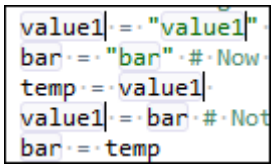
When a snippet is inserted, you can easily see all editable fields in the command line.



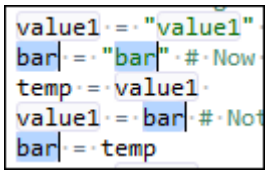
For example, in **Snippet Template**, the first editable field (**`foo`**) is highlighted.



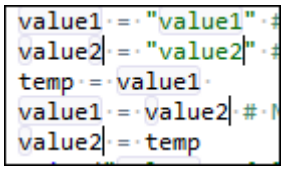
Typing something changes the **`foo`** value to whatever you type (**`value1`**). Notice that both **`foo`** values change to **`value1`** in one operation.



Pressing the **Tab** key moves the cursor to the next editable field, causing **bar** to be highlighted.



Typing something changes the **bar** value to whatever you type (**value2**). Notice that both **bar** values change to **value2** in one operation.



Pressing the **Tab** key again finalizes the code.

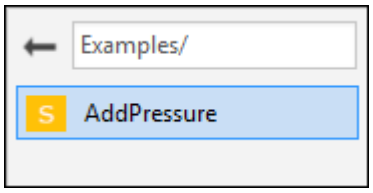
To define an editable field, you must use the following notation: **${#:*default_value*}**, where *#* is the index of the field. For the first editable field, the value is **1**. To reference an editable field that is already defined, you simply need to specify its index (**${#}**) as shown in the following figure for the snippet **Geometry**.

Snippet Inserter

The snippet inserter can be accessed in the context of the **Editor** or the **Shell**. You can use the buttons on the toolbar or the keyboard shortcut **Ctrl + i** to open the snippet inserter. It will open wherever your cursor is in the script. Once the snippet inserter is opened, you can use the mouse or the keyboard to interact with it.

Mouse Interaction

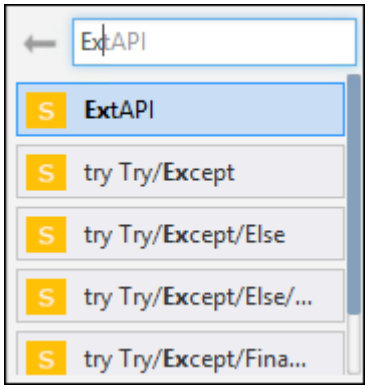
Using the mouse, you can simply click snippets to insert them into the **Editor**. If you click a snippet folder, it opens so that you can see the snippets inside it. As you browse folders, the text in the search box displays your current file path:



Clicking the back button for the search box returns you to your last location.

Keyboard Interaction

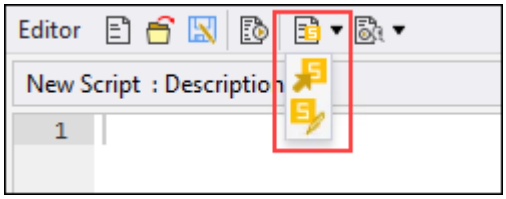
You can use the search box to search or browse snippets. As seen in the following figure, if you type **Ex**, the snippet inserter displays all snippets and snippet folders that include **Ex** in their names. From there, you can use the **Tab** or **Enter** key to select a snippet or a snippet folder.

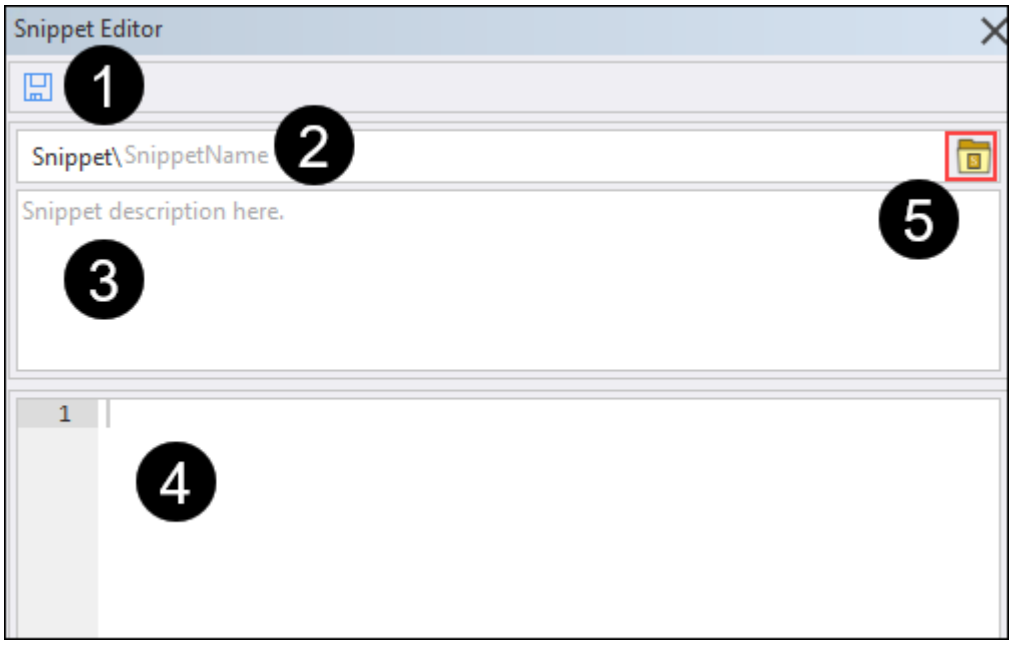


Deleting the text in the search box or using the keyboard shortcut **Alt +** ← returns you to your last location.

Snippet Creation and Management

To create snippets, you use the **Snippet Editor**, which you access from the toolbar for the **Editor**:





1. **Save button**: Saves the snippet to the path specified.
2. **Name input field**: Specifies the name or both the folder and name to which to save the snippet. Examples follow:

   - To save a snippet named **`Quick Add`** to the root folder for snippets, the name input field would look like this:

     Note:

     Root folder refers to the place where you start out when browsing snippets. For reference, the snippet **`ExtAPI`** is in the root folder.

   - To save this same snippet to the folder **`Examples`**, the name input field would look like this:

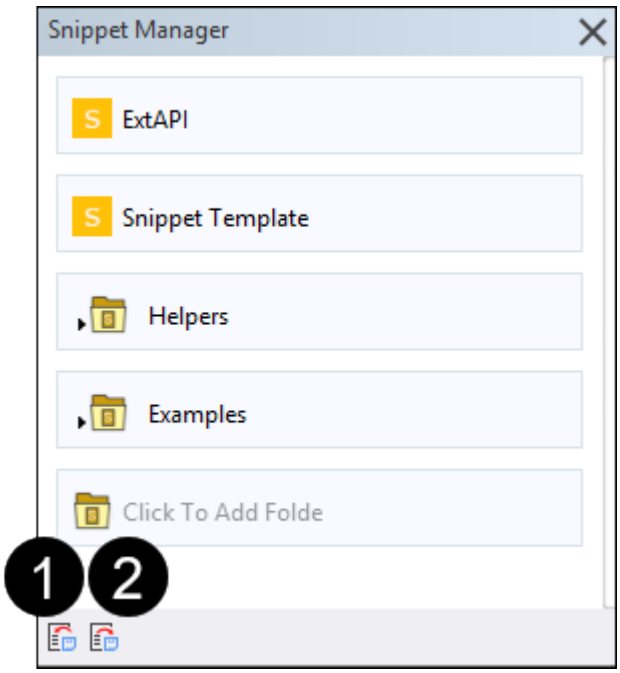
     

     Note:

     When you click an existing folder, the **Snippet Manager** inserts the path to this folder in the name input field so that the snippet will be saved to this folder. You can also manually type the path to the folder.

3. **Description**: Describes what your snippet does.
4. **Content area**: Where you write the snippet that is to be inserted when the snippet inserter is used.

5. **Snippet Manager**: Opens a view where you can manage your snippets.



Descriptions follow for the two buttons in the lower left corner of the **Snippet Manager**:

a. **Import snippet collection**: Opens a dialog box for you to select the snippet collection file. The **Snippet Manager** supports XML and JSON files. The XML files can be files that were exported using the **ACT Console**. The JSON files are snippet collections that are imported using the **Snippet Manager**. Importing snippet collection files will add the snippets that they contain to your existing snippets.

b. **Export snippet collection**: Exports all snippets in the **Snippet Manager** to a JSON file.

Note:

The **Snippet Manager** displays the **Click To Add Folder** option as the last node In every folder. You can click this option to create a new folder in the current folder.

The other options available to you in the **Snippet Manager** depend on whether you place the mouse cursor over a snippet or a snippet folder.

When you place the mouse cursor over a snippet, buttons are available for either editing or deleting the snippet:

1. **Edit snippet**: Opens the snippet in the **Snippet Editor** so that you can view and make changes to the snippet.

2. **Delete snippet**: Deletes the snippet.

> **Caution:**
>
> Deleting a snippet from the **Snippet Manager** also removes it from the snippet inserter.

When you place the mouse cursor over a folder, buttons are available for browsing the folder and either editing or deleting the folder:

1. **Browse folder**: Expand the folder to browse the snippet collection.
2. **Edit folder**: Makes the folder name editable so that you can change it.
3. **Delete folder**: Deletes the folder.

> **Caution:**
>
> Deleting a folder deletes all of its contents.

Keyboard Shortcuts

The following topics summarize the keyboard shortcuts that are available.

Editor and Shell Keyboard Shortcuts

These keyboard shortcuts are available in the **Editor**, **Shell**, or both areas.

Editor

The **Editor** supports **Ctrl + F5** as a shortcut for executing the script.

Shell

The **Shell** supports the following shortcuts:

- **Enter**: Executes the command
- **Shift + Enter**: Inserts a new line.
- **Ctrl + ↑**: Takes you to the previously executed command.

- **Ctrl +** ↓: Takes you to the next executed command.
- **Esc**: Closes autocompletion tooltips.

Editor and Shell

Both the **Editor** and **Shell** support the following shortcuts:

- **Ctl + i**: Opens the snippet inserter.
- **Ctl + Space**: Forces autocompletion to reopen.

Text Editor Keyboard Shortcuts

The following tables list keyboard shortcuts for the text editor.

Line Operation Shortcuts

Key Combination	Action
Ctrl + D	Remove line
Alt + Shift + ↓	Copy lines down
Alt + Shift + ↑	Copy lines up
Alt + ↓	Move lines down
Alt + ↑	Move lines up
Alt + Backspace	Remove to line end
Alt + Delete	Remove to line start
Ctrl + Delete	Remove word left
Ctrl + Backspace	Remove word right

Selection Shortcuts

Key Combination	Action
Ctrl + A	
Shift + ←	Select all

Key Combination	Action
Shift + →	Select left
Ctrl + Shift + ←	Select right
Ctrl + Shift + →	Select word left
Shift + Home	Select word right
Shift + End	Select line start
Alt + Shift + →	Select line end
Alt + Shift + ←	Select to line end
Shift + ↑	Select to line start
Shift + ↓	Select up
Shift + Page Up	Select down
Shift + Page Down	Select page up
Ctrl + Shift + Home	Select page down
Ctrl + Shift + End	Select to start
Ctrl + Shift + D	Select to end
Ctrl + Shift + P	Duplicate selection

Multi-Cursor Shortcuts

Key Combination	Action
Ctrl + Alt + ↑	Add multi-cursor above
Ctrl + Alt + ↓	Add multi-cursor below
Ctrl + Alt + →	Add next occurrence to multi-selection
Ctrl + Alt + ←	Add previous occurrence to multi-selection
Ctrl + Alt + Shift + ↑	Move multi-cursor from current line to the line above
Ctrl + Alt + Shift + ↓	Move multi-cursor from current line to the line below
Ctrl + Alt + Shift + →	Remove current occurrence from multi-selection and move to next
Ctrl + Alt + Shift + ←	Remove current occurrence from multi-selection and move to previous
Ctrl + Shift + L	Select all from multi-selection

Go-To Shortcuts

Key Combination	Action
Page Up	Go to page up
Page Down	Go to page down
Ctrl + Home	Go to start
Ctrl + End	Go to end
Ctrl + L	Go to line
Ctrl + P	Go to matching bracket

Folding Shortcuts

Key Combination	Action
Alt + L, Ctrl + F1	Fold selection
Alt + Shift + L, Ctrl + Shift + F1	Unfold

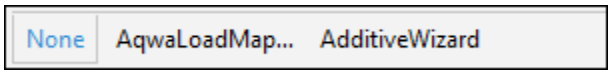
Other Shortcuts

Key Combination	Action
Tab	Indent
Shift + Tab	Outdent
Ctrl + Z	Undo
Ctrl + Shift + Y, Ctrl + Y	Redo
Ctrl + T	Transpose letters
Ctrl + Shift + U	Change to lower-case
Ctrl + U	Change to upper-case
Insert	Overwrite

Scope Selection for ACT Extensions

Each ACT extension runs Python code in its own script scope. To access global variables or functions associated with an extension, you must first select the associated tab. There will be one tab for each extension loaded in Mechanical.

The following image shows the state of the tabs when supplied extensions **`AqwaLoadMapping`** and **`AdditiveWizard`** are loaded:



Scopes are only shown if debug mode is enabled for ACT extensions. For more information, see Debug Mode in the *ACT Developer's Guide*.

Key Usage Concepts

Understanding these key concepts makes Mechanical's scripting easier to learn:

- Mechanical APIs can be used to access much of Mechanical, including tree objects and their properties.
- It is important to understand a key but subtle distinction between two APIs related to the mesh:
  - **`Model.Mesh`** accesses the object **`Mesh`** in the Mechanical tree, containing APIs for that object's properties in the **Details** view. It can be used to add and access mesh controls.
  - **`DataModel.MeshByName("Global")`** accesses FE information (node and element locations, element connectivity, and so on).
- It is important to understand a key but subtle distinction between two APIs related to the geometry:
  - **`DataModel.GeoData`** accesses the underlying geometry attached to Mechanical.
  - **`Model.Model.Geometry`** accesses the object **`Geometry`** in the Mechanical tree.
- The underlying **`Geometry`**, much like the **`Tree`**, is hierarchical. For example, **`DataModel.GeoData.Assemblies[0].Parts[0].Bodies[0].Volume`** accesses the volume for the first part in the first (and only) assembly.
- All bodies have parts as their parents, even if they are not multibody parts. This is important to understand in both **`GeoData`** and when traversing the object **`Geometry`** of the **`DataModel`**. Although a part might be hidden from the Mechanical interface, the part is always there. For more information, see Multibody Behavior and Associativity in the *ANSYS Mechanical User's Guide*.
- For more information on accessing the properties of an object, including those for traversing the geometry, mesh, simulation, and results, see Mechanical APIs (p. 29).
- It is often useful to store variables to access later rather than using the same API over and over. For example, using the following three commands is better than duplicating the **`DataModel.MeshByName("Global")`** expression in the second and third commands:

```
mesh = DataModel.MeshByName("Global")
```

```
mesh.ElementCount
```

```
mesh.NodeCount
```

- Looping is a fundamental concept in any scripting language. Within the object **`Geometry`**, you can use loops to add, modify, or export property data. The following script loops over all contacts and change their formulation to MPC:

```
contact_region_list = DataModel.GetObjectsByType(DataModelObjectCategory.ContactRegion)
```

For **contact_region** in **contact_region_list**:

```
    contact_region.ContactFormulation = ContactFormulation.MPC
```

- The object **Transaction()** can be used to speed up a block of code that modifies multiple objects. This object ensures that the tree is not refreshed and that only the bare minimum graphics and validations occur while the transaction is in scope. The object **Transaction()** should only be used around code blocks that do not require state updates, such as solving or meshing. For example, to avoid redundant work from each addition, the following example use the object **Transaction()** before code that adds many objects:

```
with Transaction():
    for bodyId in bodyIds:
    ...
```

- If you cannot find what you want, check the attribute **InternalObject**. While the ACT API has "wrapped" many useful aspects of Mechanical, it has not wrapped everything, for various reasons. Many API objects have an **InternalObject** attribute that you can use to find additional capabilities that are not formally exposed in ACT.

For more information about how to complete many types of Mechanical tasks using scripts, see the examples in these sections:

- Script Examples for Selection (p. 109)
- Script Examples for Interacting with Tree Objects (p. 117)
- Script Examples for Interacting with the Mechanical Session (p. 129)

Threading

Although using Mechanical scripting APIs on a background thread might work in practice, the APIs are generally not thread-safe, and race conditions might and often do occur when trying to access them from a background thread.

It is still possible to only run parts of your code—namely those parts that do not use these APIs and hence do not risk race conditions—on a background thread. To do so, you can offload work to a new thread. A convenient way to do this is by using the **`InvokeBackground`** method exposed on **`ExtAPI.Application`**. This method can only take a function without any arguments, but the following technique can be used to pass in arguments to that function:

```
#function that is to be run in the background. It is safe because it does not using any of the Mechanical scriptin

def gradient(vectors)
    print("Computing gradients of the vectors”)

#function that is run on the main thread
def my_script(result):
    vectors = some_function(result)
    def invokeInBackground():
        gradient(vectors)
    ExtAPI.Application.InvokeBackground(invokeInBackground)
```

Note:

A race condition is a software problem that can arise when concurrent code does not synchronize data access and mutation. These conditions are by their nature difficult to identify and reproduce, and they sometimes lead to seemingly random problems. There is no way to predict the outcome of a race condition. Alarmingly, it is possible for code to work well for years and then suddenly start to crash because of a race condition.

Additional Resources

To help you use scripts in Mechanical, many resources are available.

- The installed Python scripts that Mechanical uses can be insightful. These scripts are in your ANSYS installation directory at `.../aisol/DesignSpace/DSPages/Python`. Useful scripts include `toolbar.py` and `selection.py`. These scripts are run when using many of the options in the **Select** group on the Mechanical ribbon's **Selection** tab. For more information, see Selection Tab in the *ANSYS Mechanical User's Guide*.
- The *ACT API Reference Guide* provides descriptions of all ACT API objects, methods, and properties.

Note:

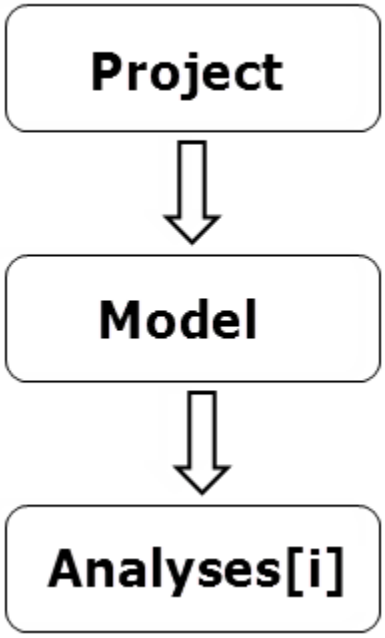
ANSYS support is available anytime by emailing <`support@ansys.com`>.

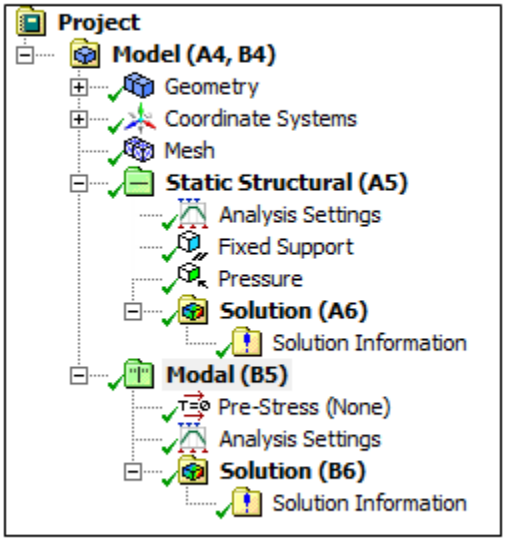
Mechanical APIs

Mechanical APIs (Application Programming Interfaces) provide access to the native functionality of ANSYS Mechanical.

Mechanical API Introduction

Using APIs, you can access, modify, and add objects in the Mechanical tree (**`Project`**, **`Model`**, **`Analysis`**, and so on).





Note:

When you add an object using the API, the default values for properties in the **Details** view in Mechanical are the same as when you add an object directly in Mechanical.

Mechanical API Notes

The following topics provide migration notes for those of you who have existing scripts for Mechanical and known issues and limitations:

Mechanical API Migration Notes

Mechanical API Known Issues and Limitations

Mechanical API Migration Notes

As improvements are made to Mechanical APIs and the way that they display and transmit data, great efforts are taken to ensure that changes are backwards-compatible. For your convenience, this section lists 2020 R1 Mechanical API changes that might impact your existing scripts so that you can determine if any action is necessary before migrating them.

Note:

For general ACT migration information, see Migration Notes in the *ACT Developer's Guide*.

Change to discrete values for input variables

When setting discrete values for an input variable, the values must be in a sorted order. Otherwise, an exception is thrown. In previous releases, values also had to be in a sorted order, but no exception was thrown. For more information, see Setting Discrete Values for Variables (p. 57).

Deprecated method for Condensed Part objects

For Condensed Part objects, the method **`GenerateCondensedGeometry`** has been deprecated. Matching the GUI option, the new method **`GenerateCondensedParts`** should be used instead. Currently, using the deprecated method will still generate the Condensed Part object, but a message will display, indicating that you should be using the new method.

Property change for bearings

For bearings, the property **`ReferenceSet`** has been renamed to **`ReferenceLocation`** to match the GUI. You must now use **`ReferenceLocation`** because **`ReferenceSet`** is obsolete.

Property change for Interface Delamination and SMART Crack Growth objects

For Interface Delamination and SMART Crack Growth objects, the **`enum`** type for the property **`AutomaticTimeStepping`** has changed to **`AutomaticOrManual`**.

Mechanical API Known Issues and Limitations

This section lists known issues and limitations related to Mechanical APIs.

Note:

For general ACT known issues and limitations, see Known Issues and Limitations in the *ACT Developer's Guide*.

General Issues and Limitations

ACT is unable to create a chart from ANSYS Mechanical

When using ACT to create a figure from the chart API, the following error prevents the graphics display in the Mechanical window:

```
Object reference not set to an instance of an object.
```

As a workaround, add the following code to your script to create an empty window in which the chart can display:

```
import clr
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
import Ansys.UI.Toolkit
if Ansys.UI.Toolkit.Window.MainWindow == None:
   Ansys.UI.Toolkit.Window.MainWindow = Ansys.UI.Toolkit.Window()
```

Limitations on ACT Postprocessing of Mechanical Results

- **Scoping for custom results defined in ACT extensions not supported**

  Custom results do not support using a geometric path as scoping. You can only use a selection of nodes and elements as scoping.

- **Scoping limitation for results using external solvers**

  Results using external solvers cannot be scoped to **`ElemNodal`** locations. This is not currently supported.

- **Compressed result file not supported**

  The ACT postprocessing API does not support the compressed result file for Mechanical, which is created by the Mechanical APDL command **`/FCOMP`**.

- **A node merge action on the mesh is unsupported**

  The ACT postprocessing API does not support a node merge action on the mesh.

- **ShellPosition command on shell bodies is unsupported**

  The ACT postprocessing API does not account for the ShellPosition command for dealing with shell bodies.

- **Orientation nodes on beam bodies are not discarded**

  The ACT postprocessing API returns the results on orientation nodes when dealing with beam bodies, even though these results are irrelevant.

Mechanical Limitations Unique to Running on Linux

- **Some property controls are not supported**

  Mechanical does not support the following property controls on Linux:

  – **`FileOpen`**

  – **`FolderOpen`**

  – **`PropertyTable`**

  These controls are implemented using the ANSYS UI Toolkit, which is currently not supported on Linux when executed within Mechanical.

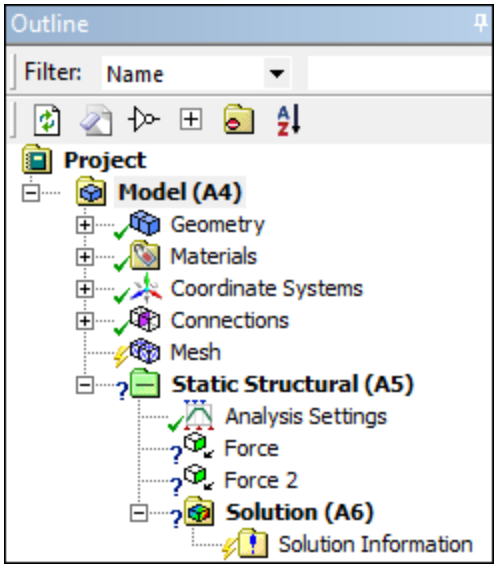
- **Graphics API issues in Mechanical when no extensions are loaded**

  When no extensions are loaded, there are some limitations on the **`Graphics`** API from the **ACT Console** in Mechanical (and also in ANSYS DesignModeler). For instance, Factory2D does not work. Therefore, you should load one or more extensions before using the **`Graphics`** API from the **ACT Console**.

Object Access

Using APIs, you can access Mechanical tree objects. The object representing the project node of the tree is **`DataModel.Project`**.

Each object can have children that you access by using the property **`Children`**. For example, assume that the **Outline** view in Mechanical looks like this:



To return all objects directly under the **Project** node, you enter this:

```
DataModel.Project.Children
```

Under the **Project** node is the **Model** node. Some examples follow for accessing **Model** child nodes:

```
Mesh = Model.Mesh
```

```
Connections = Model.Children[3]
```

To access all objects with a given name, you use the method **`GetObjectsByName`**:

```
DataModel.GetObjectsByName("name")
```

To access all objects if a given type, you use the method **`GetObjectsByType`** and pass the data model object category (such as **`DataModelObjectCategory.Force`**) as an argument:

```
DataModel.GetObjectsByType(DataModelObjectCategory.Force)
```

Property Types

Each object in the tree might have properties that display in the **Details** view. The APIs use a handful of types to refer to different kinds of properties. Descriptions and examples of the most common types follow.

Quantity: A real value with a unit typically related to a physical quantity.

```
my_object.ElementSize = Quantity("0.1 [m]")
```

Number (float and integer): A real or integer value.

```
my_object.TetraGrowthRate = 2
```

Boolean: Either **`True`** or **`False`**.

```
my_object.WriteICEMCFDFiles = True
```

Enumeration: A named value out of a set of possible named values.

```
mmesh_method.Algorithm = MeshMethodAlgorithm.PatchIndependent
```

Entity Selection ((**`ISelectionInfo`** or **`IMechanicalSelectionInfo`**): A value representing a geometric or mesh selection. Some objects are valid selections (such as a Named Selection object) and can be used as values for these properties. To apply a list of entities directly, you can:

- Create an instance of **`MechanicalSelectionInfo`** and specify the IDs.
- Assign this instance to an entity selection property (such as **`Location`**).

```
my_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
my_selection.Ids= [28,25]

My_object.Location = my_selection
```

Tree

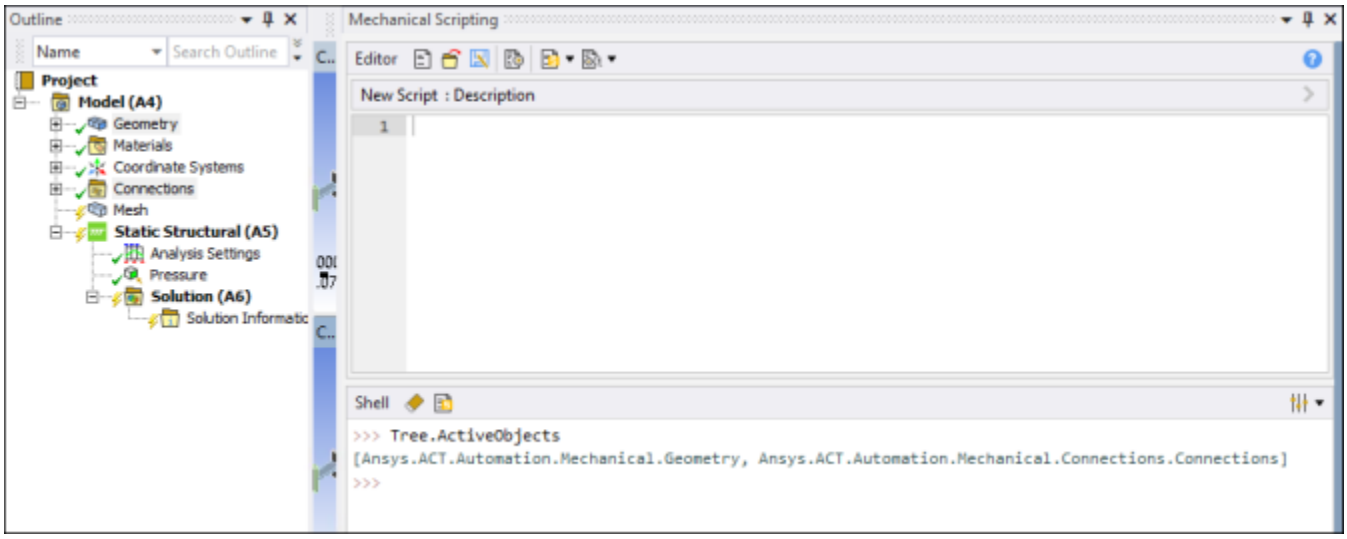
The **`Tree`** object provides access to the Mechanical tree.

To access this object, simply enter **`Tree`**. The following table provides a sampling of the APIs available on this object. Some usage examples appear after the table. For a comprehensive listing of methods and properties of the **`Tree`**, see Tree in the *ACT Online API and XML Reference Guide*.

Member	Description
`ActiveObjects`	Lists all selected objects. Read-only.
`AllObjects`	Lists all of the objects available in the tree. Read-only.
`GetPathToFirstActiveObject`	Shows the full statement that must be typed to get the selected object.
`Refresh`	Refreshes the tree.

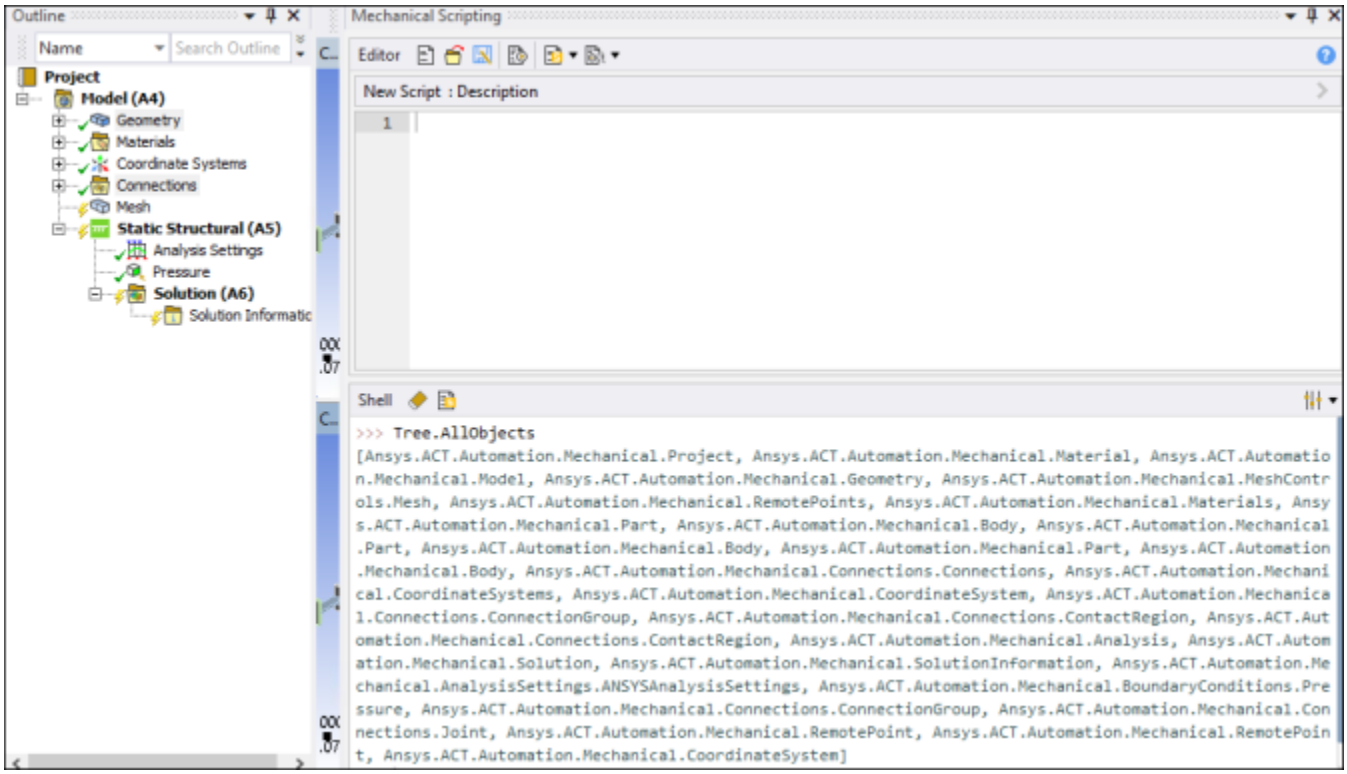
To access all objects selected in the tree:

```
Tree.ActiveObjects
```



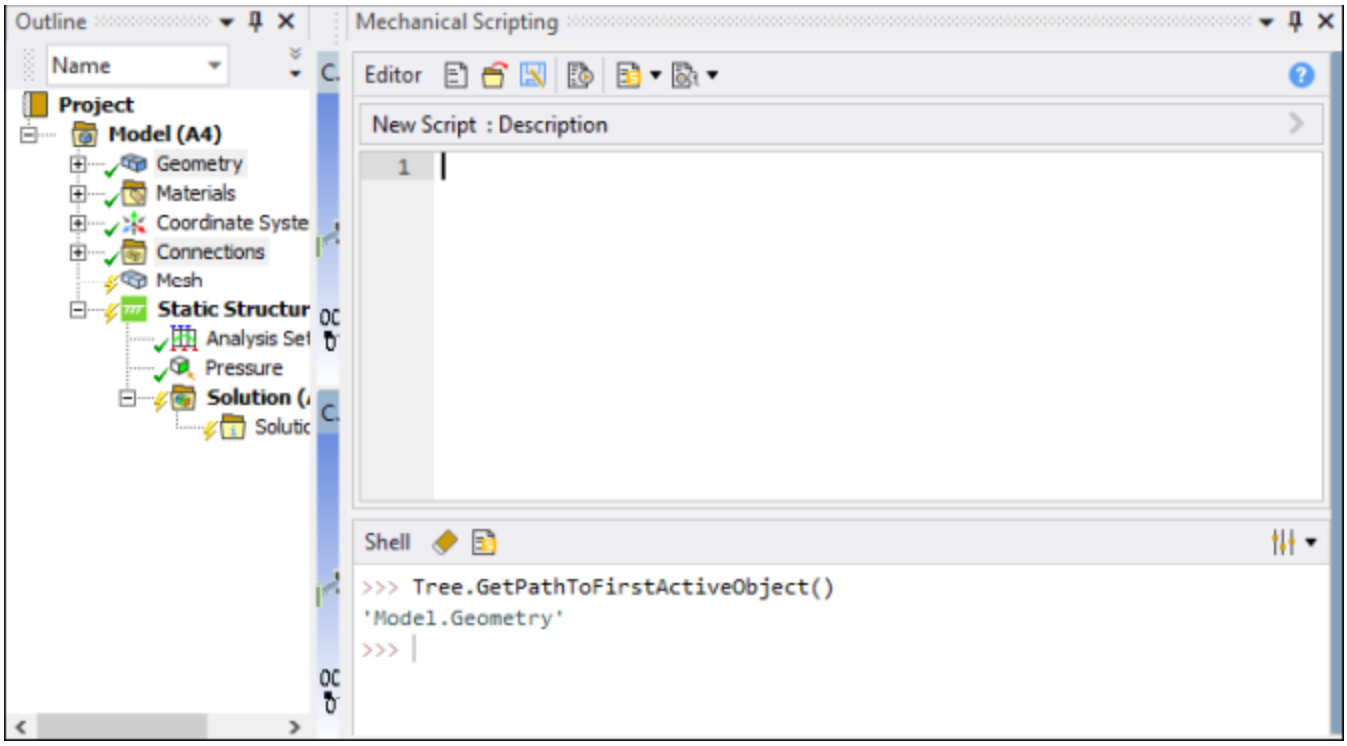
To access all objects in the tree:

```
Tree.AllObjects
```



To access the first object selected in the tree:

```
Tree.GetPathToFirstActiveObject()
```



Here are some other helpful APIs for performing tasks in the tree:

- Iterating

```
for obj in Tree
```

- Sort

```
Tree.Sort()

Tree.ClearSort()
```

- Filter

```
Tree.Filter(tag="tagname")

Tree.Filter(state=ObjectState.Suppressed)

Tree.Filter(visibility=False)

Tree.ClearFilter
```

- Find

```
objects = Tree.Find(name="substring", state=ObjectState.Unsuppressed)

for obj in objects:
```

- Events

```
def myFunc(sender, args):...
```

```
Tree.OnActiveObjectChanged += myFunc
```

- Grouping

```
Tree.Group([obj1, obj2, obj3, ...])

Tree.HideAllGroupingFolders()

Tree.ShowAllGroupingFolders()
```

- Multi-select

```
Tree.Activate([obj1, obj2, obj3, ...])
```

Model Objects

The following topics describe the objects **`Geometry`**, **`Mesh`**, **`Connections`**, and **`Analysis`** and how you can access and manipulate them:

Accessing and Manipulating the Geometry Object

The **`Geometry`** object provides an API for the **Geometry** tree object.

To access the **`Geometry`** object:

```
geometry = Model.Geometry
```

The **`Geometry`** object exposes several convenient methods for adding child objects. For example, you can add a point mass to the **`Geometry`** object by calling the method **`AddPointMass`**.

```
point_mass = geometry.AddPointMass()
```

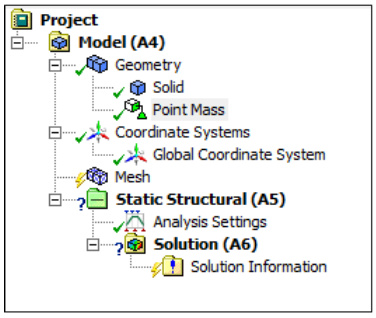
For this point mass, you can assign a location:

```
my_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
my_selection.Ids = [22]
point_mass.Location = my_selection
```

For the above point mass, accessible properties include:

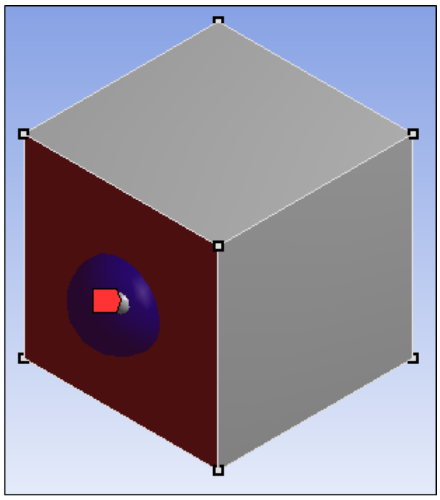
```
point_mass.Mass = Quantity("12 [kg]")
point_mass.MassMomentOfInertiaX = Quantity("1.1 [kg m m]")
point_mass.MassMomentOfInertiaY = Quantity("1.2 [kg m m]")
point_mass.MassMomentOfInertiaZ = Quantity("1.3 [kg m m]")
point_mass.Behavior = LoadBehavior.Coupled
point_mass.PinballRegion = Quantity("0.2 [m]")
```

Combining the three previous actions, the geometry now contains a fully defined point mass.



Details of "Point Mass"

Scope	
Scoping Method	Geometry Selection
Applied By	Remote Attachment
Geometry	1 Face
Coordinate System	Global Coordinate System
X Coordinate	0.5 m
Y Coordinate	0.5 m
Z Coordinate	1. m
Location	Click to Change
Definition	
Mass	12. kg
Mass Moment of Inertia X	1.1 kg·m²
Mass Moment of Inertia Y	1.2 kg·m²
Mass Moment of Inertia Z	1.3 kg·m²
Suppressed	No
Behavior	Coupled
Pinball Region	0.2 m



You can export the **`Geometry`** object to an STL (STereoLithography) file, which is the most commonly used file format in 3D printing. The following command exports the geometry object to an STL file.

```
Model.Geometry.ExportToSTL("C:\Temp\geoasstl.stl")
```

The result is the creation of a geometry file (`geoasst1.stl`) to the fully qualified directory path (`C:\Temp`).

Accessing and Manipulating the Mesh Object

The **`Mesh`** object provides an API for the **Mesh** tree object.

To access the **`Mesh`** object:

```
mesh = Model.Mesh
```

The **`Mesh`** object exposes several convenient methods to add mesh controls. For example, you can create a meshing control that applies a patch-independent algorithm to the mesh by calling the method **`AddAutomaticMethod`**.

```
mesh_method = mesh.AddAutomaticMethod()
```

`Mesh control` objects often require a valid scoping. You can satisfy this requirement by setting the **`Location`** property:

```
my_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
my_selection.Ids = [16]
mesh_method.Location = my_selection
```

For the above mesh control, accessible properties include:

```
mesh_method.Method = MethodType.AllTriAllTet
mesh_method.Algorithm = MeshMethodAlgorithm.PatchIndependent
mesh_method.MaximumElementSize = Quantity("0.05 [m]")
mesh_method.FeatureAngle = Quantity("12.000000000000002 [degree]")
mesh_method.MeshBasedDefeaturing = True
mesh_method.DefeaturingTolerance = Quantity("0.0001 [m]")
mesh_method.MinimumSizeLimit = Quantity("0.001 [m]")
mesh_method.NumberOfCellsAcrossGap = 1
```

```
mesh_method.CurvatureNormalAngle = Quantity("36 [degree]")
mesh_method.SmoothTransition = True
mesh_method.TetraGrowthRate = 1
```

Accessing and Manipulating the Connections Object

The **`Connections`** object provides an API for the **Connections** tree object.

To access the **`Connections`** object:

```
connections = Model.Connections
```

The **`Connections`** object exposes several convenient methods for adding connections. For example, you can add a beam or set the contact type to frictionless on one of the model's contact regions:

```
beam = connections.AddBeam()

contact_region = connections.Children[0].Children[0]
contact_region.ContactType = ContactType.Frictionless
```

`Beam` objects require a valid scoping. You can satisfy this requirement by setting the appropriate property. For a beam, you set the **`ReferenceLocation`** and **`MobileLocation`** properties:

```
reference_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
reference_scoping.Ids = [110]
beam.ReferenceLocation = reference_scoping
mobile_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
mobile_scoping.Ids = [38]
beam.MobileLocation = mobile_scoping
```

For the above beam, accessible properties include:

```
beam.ReferenceBehavior = LoadBehavior.Deformable
beam.ReferencePinballRegion = Quantity("0.001 [m]")
beam.Radius = Quantity("0.005 [m]")
beam.MobileZCoordinate = Quantity("6.5E-03 [m]")
beam.MobilePinballRegion = Quantity("0.001 [m]")
```

Accessing and Manipulating the Analysis Object

The **`Analysis`** object provides an API for the **Environment** tree object.

To access the first **`Analysis`** object in the tree and its settings:

```
analysis1 = Model.Analyses[0]
analysis_settings = analysis1.AnalysisSettings
```

The **`Analysis`** object exposes several convenient methods. For example, you can add boundary conditions like bolt pretensions, loads, and fixed supports:

```
bolt = analysis1.AddBoltPretension()
pressure = analysis1.AddPressure()
force = analysis1.AddForce()
support = analysis1.AddFixedSupport()
```

Boundary condition objects often require a valid scoping. You can satisfy this requirement by setting **Location** properties:

```
pressure_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
pressure_scoping.Ids = [220]
pressure.Location = pressure_scoping
force_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
force_scoping.Ids = [219]
force.Location = force_scoping
```

For the above boundary conditions, accessible properties include:

```
bolt.SetDefineBy(1, BoltLoadDefineBy.Load) # Change definition for step #1.
bolt.Preload.Output.SetDiscreteValue(0, Quantity("15 [N]")) # Change preload value for step #1.
pressure.Magnitude.Output.Formula = '10*time' # To use a formula
pressure.Magnitude.Output.DiscreteValues=[Quantity('6 [Pa]')] # To use a direct value
force.Magnitude.Output.DiscreteValues=[Quantity('11.3 [N]'), Quantity('12.85 [N]')]
```

Object Traversal

The following topics describe how to traverse the objects **Geometry**, **Mesh**, and **Results**:

Traversing the Geometry

Traversing the Mesh

Traversing Results

Note:

- The sample code in this section is taken from the supplied extension `TraverseExtension`. You can download the package of extension examples from the developer help panel for the ACT Start Page.
- For comprehensive information on interfaces and properties, see the *ANSYS ACT API Reference Guide*.

Traversing the Geometry

The APIs for geometry data provide access to the underlying geometry that is attached in Mechanical. For example, the API could be used to determine the faces associated with an edge.

The basic hierarchy of the geometry is:

```
- Geometry
  - Assembly
    - Part
      - Body
        - Shell
        - Face
        - Edge
        - Vertex
```

An example Python function to traverse geometry data follows. Here, an object of type **IGeoData** is obtained from the object **Analysis** using the property **GeoData**. The object **GeoData** is then used to access the list of **IGeoAssembly** with the property **Assembly**. For each of the objects in

this list, the property **`Parts`** is used to access the list of **`IGeoPart`** objects on the assembly. This pattern is repeated through the hierarchy of the geometry down to the vertices of each edge.

```
def traversegeometry(analysis):
    now = datetime.datetime.now()
    f = open("C:\\geoDump.txt",'w')
    f.write("*.*.*.*.*.*.*.*\n")
    f.write(str(now)+"\n")
    # --- IGeometry Interface
    # +++ Properties and Methods
    # +++    Assemblies
    # +++    CellFromRefId
    # +++    SelectedRefIds
    geometry = analysis.GeoData
    assemblies = geometry.Assemblies
    assemblies_count = assemblies.Count
    # --- IGeoAssembly Interface
    # +++ Properties and Methods
    # +++    Name
    # +++    Parts
    for assembly in assemblies:
        assembly_name = assembly.Name
        parts = assembly.Parts
        parts_count = parts.Count
        # --- IGeoPart Interface
        # +++ Properties and Methods
        # +++    Name
        # +++    Bodies
        for part in parts:
            part_name = part.Name
            bodies = part.Bodies
            bodies_count = bodies.Count
            # --- IGeoBody Interface
            # +++ Properties and Methods
            # +++    Name
            # +++    Vertices
            # +++    Edges
            # +++    Faces
            # +++    Shells
            # +++    Material
            for body in bodies:
                faces = body.Faces
                faces_count = faces.Count
                # --- IGeoFace Interface
                # +++ Properties and Methods
                # +++    Body
                # +++    Shell
                # +++    Vertices
                # +++    Edges
                # +++    Loops
                # +++    Area
                # +++    SurfaceType
                # +++    PointAtParam
                # +++    PointsAtParams
                for face in faces:
                    edges = face.Edges
                    edges_count = edges.Count
                    # --- IGeoEdge Interface
                    # +++ Properties and Methods
                    # +++    Faces
                    # +++    Vertices
                    # +++    StartVertex
                    # +++    EndVertex
                    # +++    Length
                    # +++    CurveType
                    # +++    Extents
                    # +++    IsParamReversed
                    # +++    ParamAtPoint
                    # +++    PointAtParam
                    # +++    PointsAtParams
                    for edge in edges:
```

```
                        vertices = edge.Vertices
                        vertices_count = vertices.Count
                        # --- IGeoVertex Interface
                        # +++ Properties and Methods
                        # +++ Edges
                        # +++ Faces
                        # +++ Bodies
                        # +++ X
                        # +++ Y
                        # +++ Z
                        for vertex in vertices:
                            xcoord = vertex.X
                            ycoord = vertex.Y
                            zcoord = vertex.Z
                            try:
                                f.write("      Vertex: "+vertex.ToString()+", X = "+xcoord.ToString()+", Y = "+y
                            except:
                                continue
    f.close()
    return
```

Traversing the Mesh

The API for mesh data provides access to the underlying mesh in Mechanical.

An example Python function to traverse mesh data follows. Here, an object of type **`IMeshData`** is obtained from the object **`Analysis`** using the property **`MeshData`**. The object **`IMeshData`** is then used to access the list of element IDs with the property **`Elements`**. Using each of the element IDs in the returned list, the method **`ElementbyId`** is called to access the element data with **`IElement`**. This pattern is repeated for all of the nodes for each element. Finally, the coordinates of the nodes are queried.

```
import datetime
def traversemesh(analysis):
    now = datetime.datetime.now()
    f = open("C"\\MeshDump.txt",'w')
    f.write("*.*.*.*.*.*.*.*\n")
    f.write(str(now)+"\n")
    # --- IMesh Interface
    # +++ Properties and Methods
    # +++ MeshRegion
    # +++ Node
    # +++ Element
    # +++ Nodes
    # +++ Elements
    # +++ NumNodes
    # +++ NumElements
    mesh = analysis.MeshData
    elementids = mesh.ElementIds
    # --- IElement Interface
    # +++ Properties and Methods
    # +++ Id
    # +++ Type
    # +++ Nodes
    for elementid in elementids:
        element = mesh.ElementById(elementid)
        nodeids = element.NodeIds
        # --- INode Interface
        # +++ Properties and Methods
        # +++ Id
        # +++ X
        # +++ Y
        # +++ Z
        # +++ Elements
        for nodeid in nodeids:
            node = mesh.NodeById(nodeid)
            nodex = node.X
```

```
            nodey = node.Y
            nodez = node.Z
            try:
                f.write("      Element: "+elementid.ToString()+"  Node: "+nodeid.ToString()+", X = "+nodex.ToStr
            except:
                continue
    f.close()
    return
```

Another example of traversing the mesh data follows. Only the elements of user-selected geometry entities are considered. The property **`CurrentSelection`** on the Selection Manager can be used to query the IDs of the selected geometry entities.

```
def elementcounter(analysis):
    with open("C:\\SelectedMeshEntities.txt”,'w') as f:
        geometry = analysis.GeoData
        smgr = ExtAPI.SelectionManager
        selectedids = smgr.CurrentSelection.Ids
        mesh = analysis.MeshData
        if selectedids.Count == 0:
           fwrite("Nothing Selected!")
           return
        for selectedid in selectedids:
            entity = geometry.GeoEntityById(selectedid)
            meshregion = mesh.MeshRegionById(selectedid)
            try:
                numelem = meshregion.ElementCount
                f.write("Entity of type: "+entity.Type.ToString()+
                                " contains "+numelem.ToString()+
                                " elements.")
            except:
                f.write("The mesh is empty!")
        return
```

Traversing Results

The API for direct result access allows you to do postprocessing without result objects.

An example python function to compute minimum and maximum results follows. It begins by instantiating a result reader using the method **`analysis.GetResultsData()`**. Results are retrieved relative to the finite element model and queried using either the **`elementID`** (elemental result) or the **`nodeID`** (nodal result). The displacement result **`U`** is a nodal result, whereas the stress result **`S`** is a result on nodes of the elements. The displacement result stores a set of component values for each node, where the component names are X, Y, and Z.

The function first iterates over the **`nodeIDs`** to compute the minimum and maximum values. It then iterates over the **`elementIDs`** and the nodes of each element to compute the minimum and maximum values.

Note:

The second loop over the nodes is filtered to the corner nodes of the elements because stress results are available only on these corner nodes.

Finally, the results are written to the output file.

```
def minmaxresults(analysis):
    now = datetime.datetime.now()
    f = open("C:\\resultsInfo.txt",'w')
    f.write("*.*.*.*.*.*.*.*\n")
```

```
    f.write(str(now)+"\n")
    #
    # Get the element ids
    #
    meshObj = analysis.MeshData
    elementids = meshObj.ElementIds
    nodeids = meshObj.NodeIds
    #
    # Get the results reader
    #
    reader = analysis.GetResultsData()
    reader.CurrentResultSet = int(1)
    #
    # Get the displacement result object
    displacement = reader.GetResult("U")

    num = 0
    for nodeid in nodeids:
        #
        # Get the component displacements (X Y Z) for this node
        #
        dispvals = displacement.GetNodeValues(nodeid)
        #
        # Determine if the component diplacement (X Y Z) is min or max
        #
        if num == 0:
            maxdispx = dispvals[0]
            mindispx = dispvals[0]
            maxdispy = dispvals[1]
            mindispy = dispvals[1]
            maxdispz = dispvals[2]
            mindispz = dispvals[2]

        num += 1

        if dispvals[0] > maxdispx:
            maxdispx = dispvals[0]
        if dispvals[1] > maxdispy:
            maxdispy = dispvals[1]
        if dispvals[2] > maxdispz:
            maxdispz = dispvals[2]
        if dispvals[0] < mindispx:
            mindispx = dispvals[0]
        if dispvals[1] < mindispy:
            mindispy = dispvals[1]
        if dispvals[2] < mindispz:
            mindispz = dispvals[2]

    # Get the stress result object
    stress = reader.GetResult("S")

    num = 0
    for elementid in elementids:
        element = meshObj.ElementById(elementid)
        #
        # Get the SXX stress component
        #
        stressval = stress.GetElementValues(elementid)
        #
        # Get the primary node ids for this element
        #
        nodeids = element.CornerNodeIds
        for i in range(nodeids.Count):
            #
            # Get the SXX stress component at node "nodeid"
            #
            SXX = stressval[i]
            #
            # Determine if the SXX stress component is min or max
            #
            if num == 0:
                maxsxx = SXX
```

```
                minsxx = SXX

            if SXX > maxsxx:
                maxsxx = SXX
            if SXX < minsxx:
                minsxx = SXX

            num += 1
    #
    # Write the results to the output
    #
    f.write("Max U,X:Y:Z = "+maxdispx.ToString()+" : "+maxdispy.ToString()+" : "+maxdispz.ToString()+"\n")
    f.write("Min U,X:Y:Z = "+mindispx.ToString()+" : "+mindispy.ToString()+" : "+mindispz.ToString()+"\n")
    f.write("Max SXX = "+maxsxx.ToString()+"\n")
    f.write("Min SXX = "+minsxx.ToString()+"\n")
    f.close()
```

Property APIs for Native Tree Objects

A native tree object is not defined by an ACT extension. You can use these APIs as another way to interact with properties shown in the **Details** view. For example, assume that you have stored a variable named **`rectbar`** of a **Body** object with these properties shown in the **Details** view:

Details of "rectbar"	
Graphics Properties	
Definition	
Suppressed	No
Stiffness Behavior	Flexible
Coordinate System	Default Coordinate System
Reference Temperature	By Environment
Behavior	None
Material	
Assignment	Structural Steel
Nonlinear Effects	Yes
Thermal Strain Effects	Yes
Bounding Box	
Properties	
Statistics	

To see the properties visible in the **Details** view, you enter:

```
rectbar.VisibleProperties
```

To list all properties that could be in the **Details** view for an object of this type, including those that are hidden, you enter:

```
rectbar.Properties
```

You can use property APIs to get the property's unique internal name, caption, internal value, and whether it is valid, visible, or read-only. You can also use **APIName** from the property APIs to see which API can be used directly on **rectbar** for the given **Details** view property, if an API exists.

Tip:

To change the internal value for any property, even one not yet wrapped by an API, you can specify the property's index value and then the current value and replacement value, entering something like this:

```
obj.Properties[0].InternalValue = newValue
```

For a scripting example, see Get All Visible Properties for a Tree Object (p. 118).

Details View Parameters

You can write scripts that create, query, or remove the parametrization of a property in the **Detail**s view of a native tree object. For example, in Mechanical, to the right of any property that can be parametrized, you can click the box, which causes a **P** to display, indicating that it is now a parameter.

Details of "rectbar"	
Graphics Properties	
Visible	Yes
Transparency	1
Color	
Definition	
P Suppressed	No
Stiffness Behavior	Flexible
Coordinate System	Default Coordinate System
Reference Temperature	By Environment
Behavior	None

This also adds the **Parameter Set** bar to the Workbench **Project Schematic**. Double-clicking the **Parameter Set** bar opens it so that you can see all input and output parameters in the project.

To parametrize a native property using an API, you use the method:

```
CreateParameter("PropertyName")
```

This method returns an instance of the *DetailsViewParameter* class. It can be used to query the Workbench ID for the parameter in the **Parameter Set** bar. You can also query the object and property name for which the parameter was created:

```
ID GetParameter(name)

Object GetParameter(name)

PropertyName GetParameter(name)
```

To no longer have a property be a parameter, you use the method:

```
RemoveParameter("PropertyName")
```

For a scripting example, see Parametrize a Property for a Tree Object (p. 118).

Solver Data

You can use SolverData APIs to get solver-related data specific to an object or global items applicable to the solver. Currently, **SolverData** APIs are available only for the MAPDL solver.

You use the **Solution** object to get solver data:

```
solution = Model.Analyses[0].Solution
solver_data = solution.SolverData
```

The following sections describe ways of using **SolverData** APIs to get global items and object-specific data.

Getting Global items for Solver Data

You can use **SolverData** APIs to get the maximum element ID, maximum node ID, and maximum element type for the ANSYS solver target.

- To get the maximum element ID:

  ```
  solver_data.MaxElementId
  ```

- To get the maximum node ID:

  ```
  solver_data.MaxNodeId
  ```

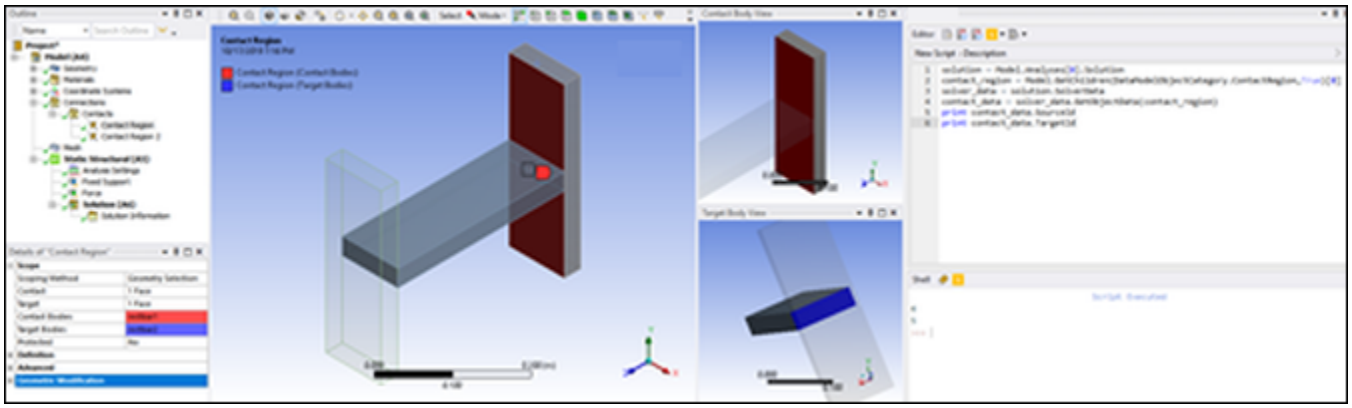
- To get the maximum element type:

  ```
  solver_data.MaxElementType
  ```

Getting Object Specific Data

You can use **SolverData** APIs to get object-related data. For example, you can use the **ContactRegion** object to get the source ID and target ID of the contact region:

```
contact_region = Model.GetChildren(DataModelObjectCategory.ContactRegion, True)[0]
contact_data = solver_data.GetObjectData(contact_region
contact_data.SourceId
contact_data.TargetId
```



You can use **`SolverData`** APIs to retrieve object data for the following objects:

- AM Support
- Beam Connection
- Bearing
- Body
- Contact Region
- Coordinate System
- Joint
- Layered Section
- Pretension Bolt Load
- Remote Point
- Spring
- Surface Coating
- Surface Load

For a scripting example, see Retrieve Object Details Using SolverData APIs (p. 120).

Note:

`SolverData` APIs are not available for objects imported using **External Model** systems.

Boundary Conditions

This section describes APIs that enable you to manipulate boundary condition fields.

Unlike many other tree objects, boundary conditions are defined using both the **Details** view and tabular data. This is because boundary conditions can be time, space, or frequency dependent. As such, the APIs use a type of object called Field. Fields provide access to both independent and dependent variables defined in the tabular data, as well as the function that can be defined in the **Details** view.

The APIs for editing boundary conditions mirror the actions that can be performed manually in the Mechanical interface. With the API, you can:

- Set tabular data, which includes setting discrete input and output values
- Set variable definition types (tabular, formula, or free)

Input and Output Variables

Inputs and outputs are represented by objects of the type Variable. For example, a force can be defined with `time` as the input variable representing the time values in seconds across the time steps defined in the analysis settings and `magnitude` as the output variable representing the magnitude of the force at those time values.

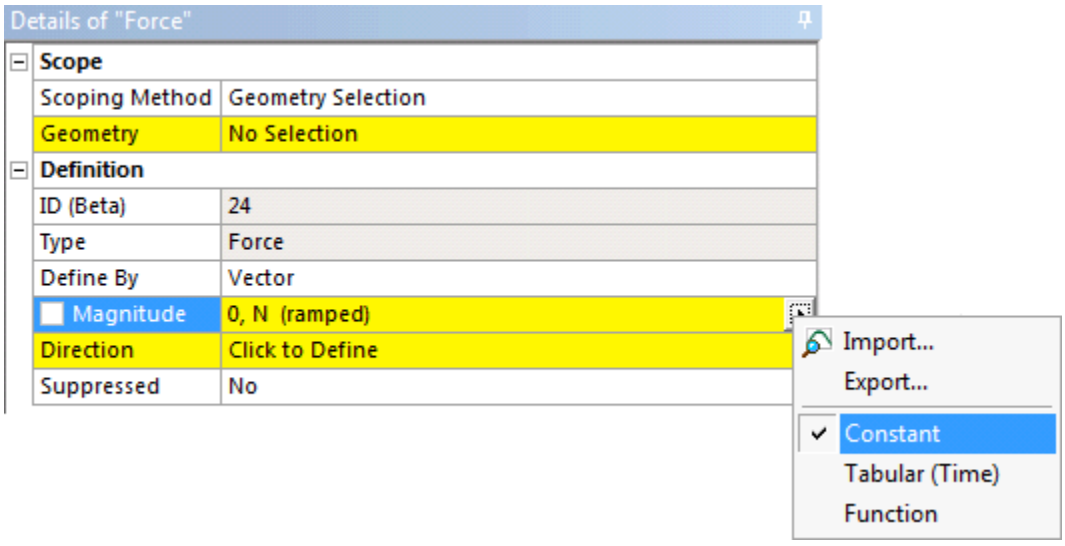
Variable Definition Types

A Field can be defined using of three variable definition types:

- **Discrete**. The variable contains a discontinuous set of values. By default, most variables are initially defined as discrete. Tabular or constant boundary conditions are considered discrete in the API.
- **Formula**. The variable is a continuous function depending on an expression, such as `"time*10"`.
- **Free**. This variable definition type is available only for certain boundary conditions such as displacements.

Setting Input Loading Data Definitions

For input variables, each variable definition type corresponds to one or more of the input loading data definition options available in the **Details** view in Mechanical. You can determine the loading data definition options available for a given input by checking the flyout menu in the user interface.



Input loading data definitions are ranked in terms of complexity:

Complexity	Variable Definition Type	Input Loading Data Definition	Description
1	Discrete	Constant (stepped or ramped)	One value if stepped, two values if ramped.
2	Discrete	Tabular	Values listed in tabular format.
3	Formula	Function	Values generated by the expression assigned to the variable.
NA	Free	NA	Indicates that the boundary condition does not add a constraint for the specified degree of freedom (DOF). (A value of 0 indicates that the DOF is constrained.)

When you use the Field API to define a boundary condition field, it automatically opts for the least complex loading data definition that is applicable to the input. For example, if a given input could be defined as **Constant (ramped)** but you execute commands defining it as **Tabular**, the API defines the input as **Constant (ramped)**.

Setting Variable Definition Types

The property **`DefinitionType`** on the variable allows you to get or set the definition types of a field when used on the output variable.

Note:

You can also change the variable definition type to **Discrete** or **Free** by using the property **`Variable.DiscreteValues`** as described in Setting Discrete Values for Variables (p. 57).

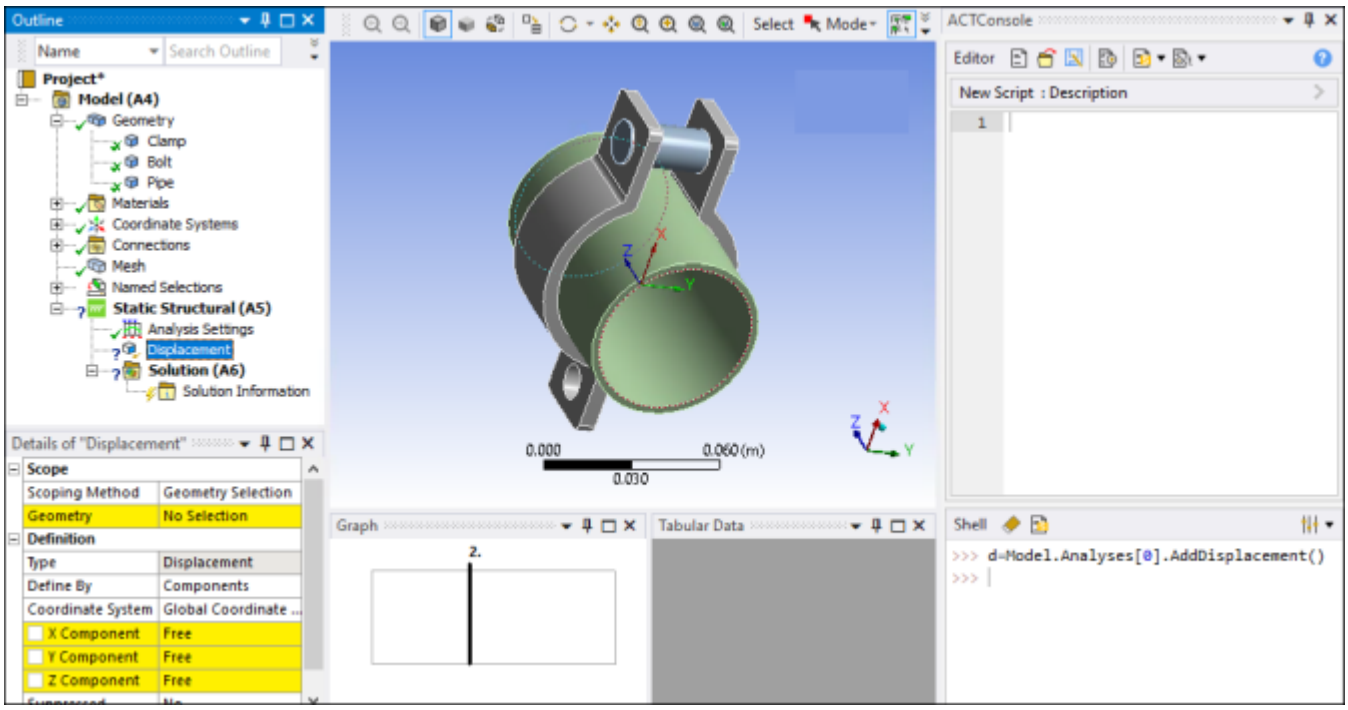
An example follows, consisting of the following steps:

Creating a Displacement and Verifying Its Variable Definition Types

You start by creating a displacement:

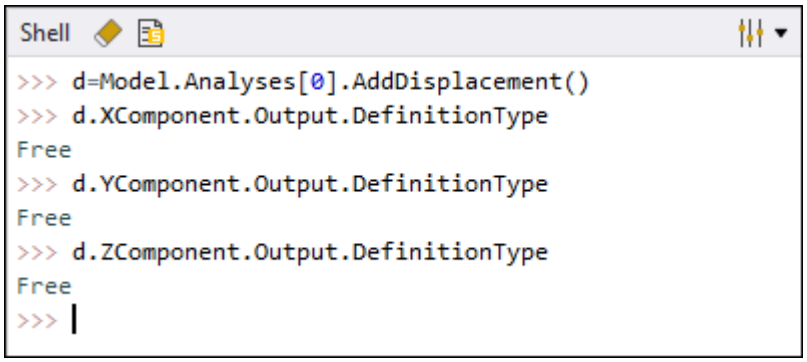
```
d=Model.Analyses[0].AddDisplacement()
```

In Mechanical, you can see that by default, the variable definition types for the displacement's X, Y, and Z components are set to **Free**.



You can verify this by executing the following commands one at a time.

```
d.XComponent.Output.DefinitionType

d.YComponent.Output.DefinitionType

d.ZComponent.Output.DefinitionType
```



Note:

Even though the expression **`d.XComponent.Output`** would produce the same output in the console when the component is **`Free`**, be aware that output is an instance of the **`Variable`** class. This object is then converted into a string by the console. Appending **`.DefinitionType`** to the expression yields the actual enum value.

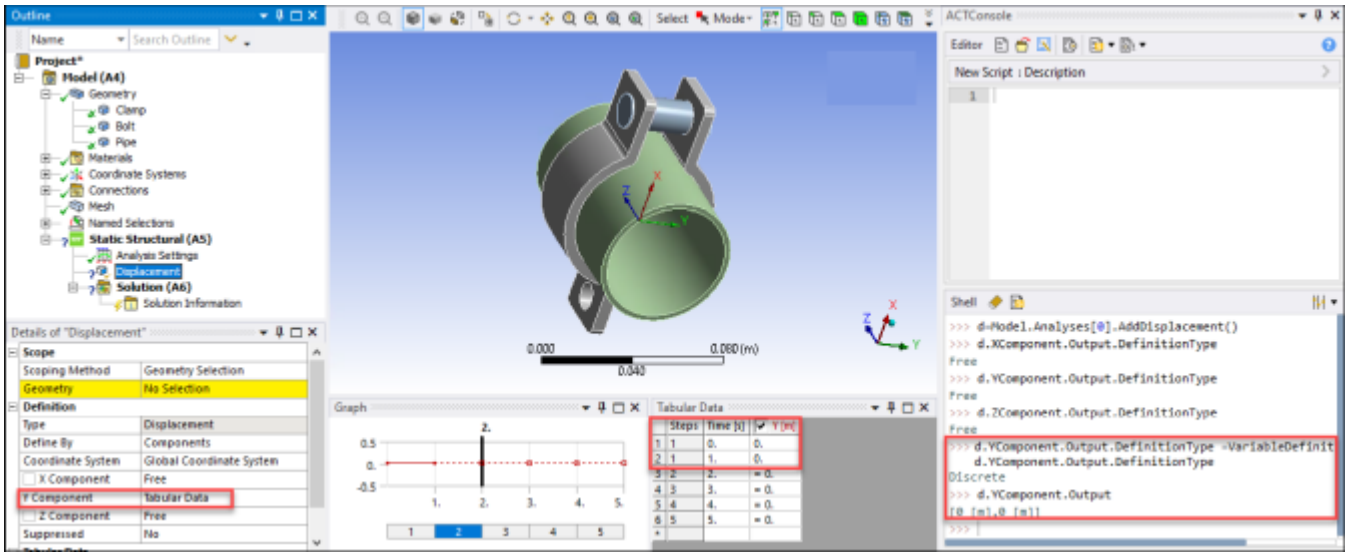
Changing the Y Component from Free to Discrete

Next, you change the variable definition type for the Y component from **Free** to **Discrete** and then access the output values:

```
d.YComponent.Output.DefinitionType =VariableDefinitionType.Discrete
d.YComponent.Output.DefinitionType


d.YComponent.Output
```

In the figure that follows, you can see that:

- The **Y Component** is set to **Tabular Data** because tabular data is the least complex discrete loading data definition that is applicable to this input.
- The **Tabular Data** window is now populated with output values of **0** and **0**.
- In the **Shell**, you can verify that the variable definition type is set to **Discrete**.
- In the **Shell**, you can verify that the output values are **0** and **0**.

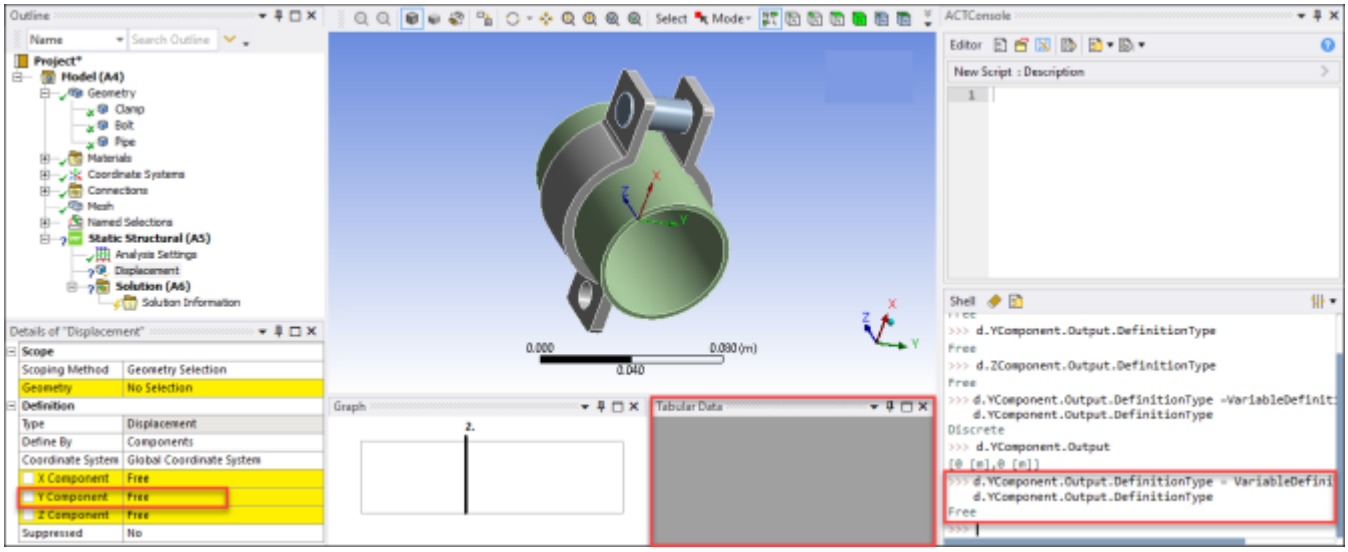


Changing the Y Component Back to Free

Finally, you change the variable definition type for the Y output from **Discrete** back to **Free**:

```
d.YComponent.Output.DefinitionType = VariableDefinitionType.Free
d.YComponent.Output.DefinitionType
```

In the following figure, you can see that the **Y Component** is set back to **Free** and the **Tabular Data** window is now empty.



Setting Discrete Values for Variables

The property **`DiscreteValues`** on the variable allows you to get and set the input and output variable values of a field:

Controlling the Input Variable

Controlling the Output Variable

Controlling the Input Variable

The list content provided to an input variable determines if it is set to a constant value or to **Tabular Data**. If the list contains only one quantity, the input is set to this constant value. If the list contains multiple quantities, the input is set to **Tabular Data**.

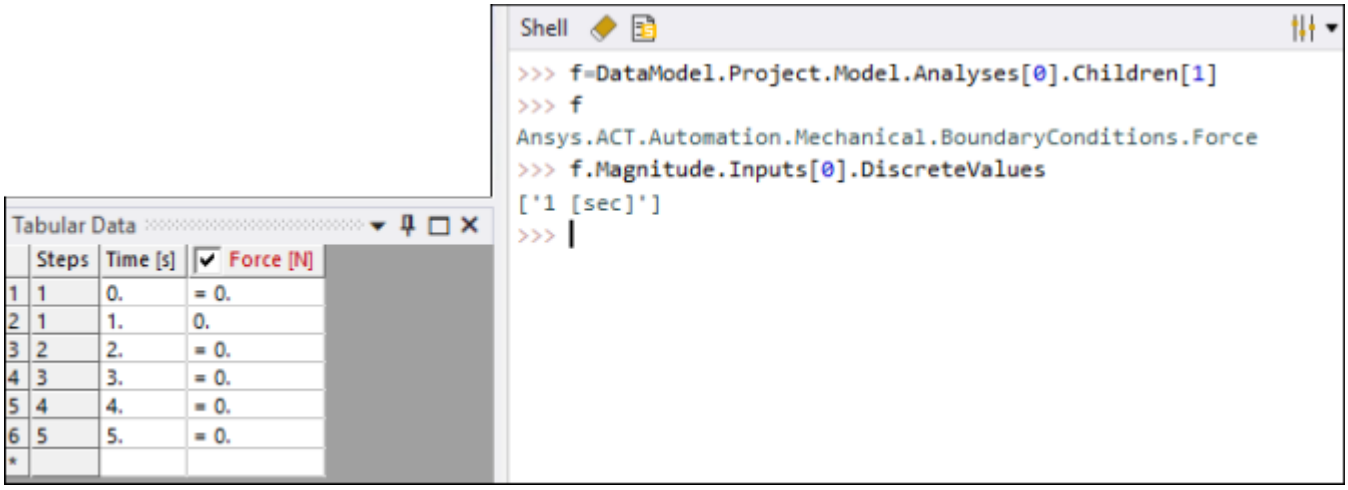
Note:

Input values for a discrete variable must be in a sorted order as shown in this example:

```
Pressure.Magnitude.Inputs[0].DiscreteValues = [Quantity("0 [s]"), Quantity("1 [s]"), Quantity("2 [s]"), Qua
```

Getting Discrete Input Values

You can get the discrete input values for a given input variable. For example, a force in a five-step analysis looks similar to the following figure by default. The six rows in the **Tabular Data** window correspond to the discrete values of the input variable, **`time`**, with **0** seconds as the start time of the first step and **5** seconds as the end time of the last step.



Setting Discrete Input Values

You can change discrete values for a given input variable. For example, the following code sample adds a discrete value, inserting a value of **0.5** seconds without defining an additional step in the analysis:

```
f.Magnitude.Inputs[0].DiscreteValues = [Quantity("0[sec]"),Quantity("0.5[sec]"),Quantity("1[sec]")]
```

In the figure that follows, you can see that:

- A new row exists for **0.5** seconds.
- The output cells for rows 3 through 7 have a yellow background. This is because values have not been set for the output variable.

Tabular Data

	Steps	Time [s]	Force [N]
1	1	0.	0.
2	1	0.5	0.
3	1	1.	
4	2	2.	
5	3	3.	
6	4	4.	
7	5	5.	
*			

Removing Discrete Values

You can also remove a discrete value, deleting a row by defining a shorter list of values for the input variable. In the line of code that follows, the list specifies a single value, indicating that the other rows should be removed:

```
f.Magnitude.Inputs[0].DiscreteValues = [Quantity("0[sec]")]
```

In the **Tabular Data** window, only six rows now show once again. The value **=0** is actually a repeat of the value in the first row and does not correspond to an actual value stored for the variable. This is because the tabular data in Mechanical always displays a row for time values that correspond to step end times.

Tabular Data

	Steps	Time [s]	Force [N]
1	1	0.	0.
2	1	1.	= 0.
3	2	2.	= 0.
4	3	3.	= 0.
5	4	4.	= 0.
6	5	5.	= 0.
*			

Controlling the Output Variable

Output variables are not limited to the **Discrete** variable definition type, so the behavior when getting and setting outputs is slightly different from that of inputs.

Getting Discrete Output Values

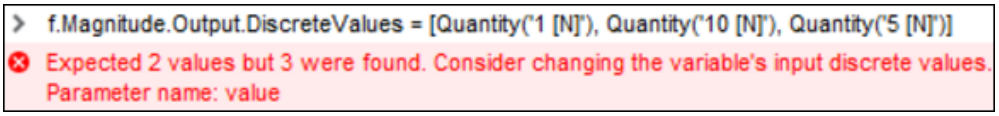
The following table shows the **`get`** behavior for the different variable definition types.

Variable Definition Type	Behavior
Discrete	Same as described earlier for getting discrete input values.
Free	For C#, the property returns **`null`**. For Python, the property returns **`None`**.
Formula	The property returns the series of evaluated variable values.

Setting Discrete Output Values

The following table shows the **`set`** behavior according to the list content provided to the property.

List Content	Behavior
None or null	The variable is switched to the **Free** definition type if it is supported by the boundary condition. Otherwise, an error is issued.

List Content	Behavior
Single quantity object	The variable is switched to the **Constant** definition type if supported in the interface. Otherwise, **Constant (ramped)** can be chosen. If neither of those two types are supported, **Tabular Data** is the default.
Two quantity objects, with the first value being 0	The variable is switched to **Constant (ramped)** if supported in the interface. Otherwise, **Tabular Data** is the default.
As many quantity objects as discrete values for the input variables	The variable is switched to **Tabular Data**.
Does not fall into any of the above categories	An error is issued, indicating that you should either: • Provide the appropriate number of values. • Change the number of input values first, as the input value count is used to determine the row count of the tabular data. Output values must then comply with this number of rows. The following figure shows a sample message. 

Setting Variable Definition Type by Acting on Discrete Values

The variable definition type can also be modified by setting the **`DiscreteValues`** property on the variable. To set a variable to a particular variable type, you set the values that are consistent with that variable type. Examples follow:

- To set an input variable to **Tabular Data**, assign a list of discrete values:

```
d.XComponent.Output.DiscreteValues = [Quantity("1 [m]"), Quantity("10 [m]")]
```

- To set a variable to **Free**, assign **`None`** to its **`DiscreteValues`**:

```
d.XComponent.Output.DiscreteValues = None
```

Adding a Load

You use the object Analysis to add a load. This topic describes adding loads to a static structural analysis.

To access the first analysis in the Mechanical project:

```
static_structural = Model.Analyses[0]
```

To change the number of steps of the analysis:

```
analysis_settings = static_structural.AnalysisSettings.NumberOfSteps = 4
```

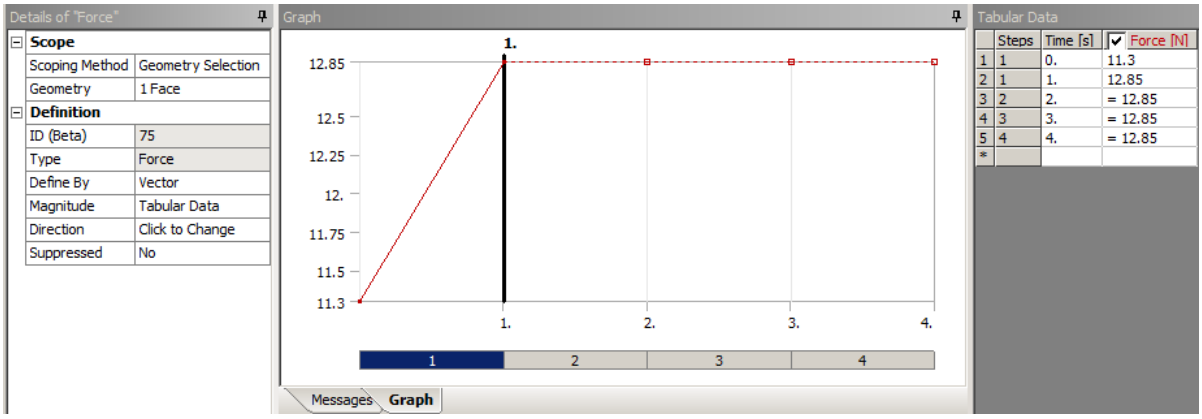
To add and define a bolt and fixed support:

```
bolt = static_structural.AddBoltPretension()
bolt_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
bolt_scoping.Ids = [200]
bolt.Location = bolt_scoping
bolt.SetDefineBy(1, BoltLoadDefineBy.Load) # Change definition for step #1.
bolt.Preload.Output.SetDiscreteValue(0, Quantity("15 [N]")) # Change preload value for step #1.

support = static_structural.AddFixedSupport()
support_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
support_scoping.Ids = [104]
support.Location = support_scoping
```

To add and define the external and internal pressures exerted on the pipe and the force on a section of the pipe:

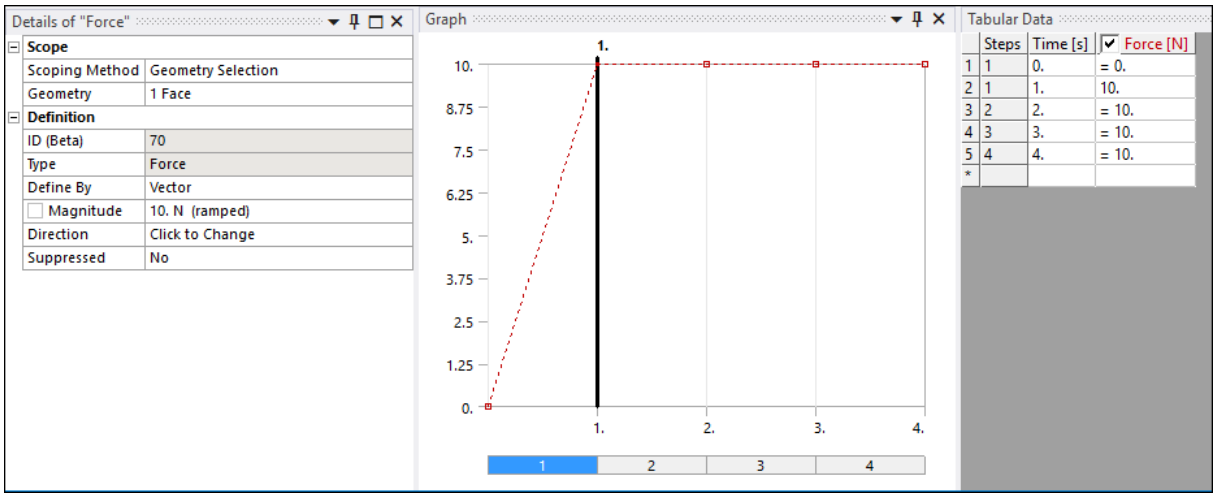
```
pressure = static_structural.AddPressure()
pressure_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
pressure_scoping.Ids = [220]
pressure.Location = pressure_scoping
pressure.Magnitude.Output.Formula = '10*time'

pressure = static_structural.AddPressure()
pressure_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
pressure_scoping.Ids = [221]
pressure.Location = pressure_scoping
pressure.Magnitude.Output.DiscreteValues=[Quantity('6 [Pa]')]

force = static_structural.AddForce()
force_scoping = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
force_scoping.Ids = [219]
force.Location = force_scoping
force.Magnitude.Output.DiscreteValues=[Quantity('11.3 [N]'), Quantity('12.85 [N]')]
```

Script execution results in the following force definition:



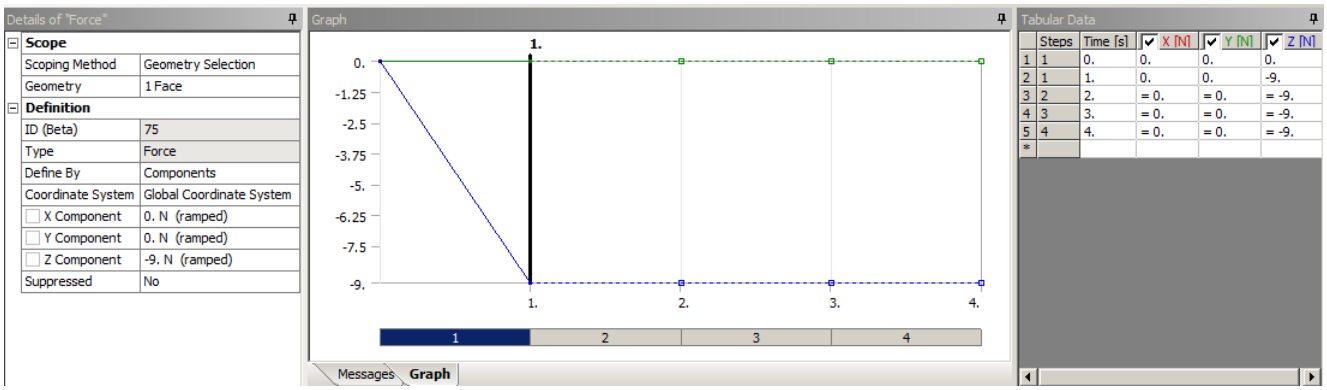
To define a constant value of the force:

```
force.Magnitude.Output.DiscreteValues=[Quantity('10 [N]')]
```



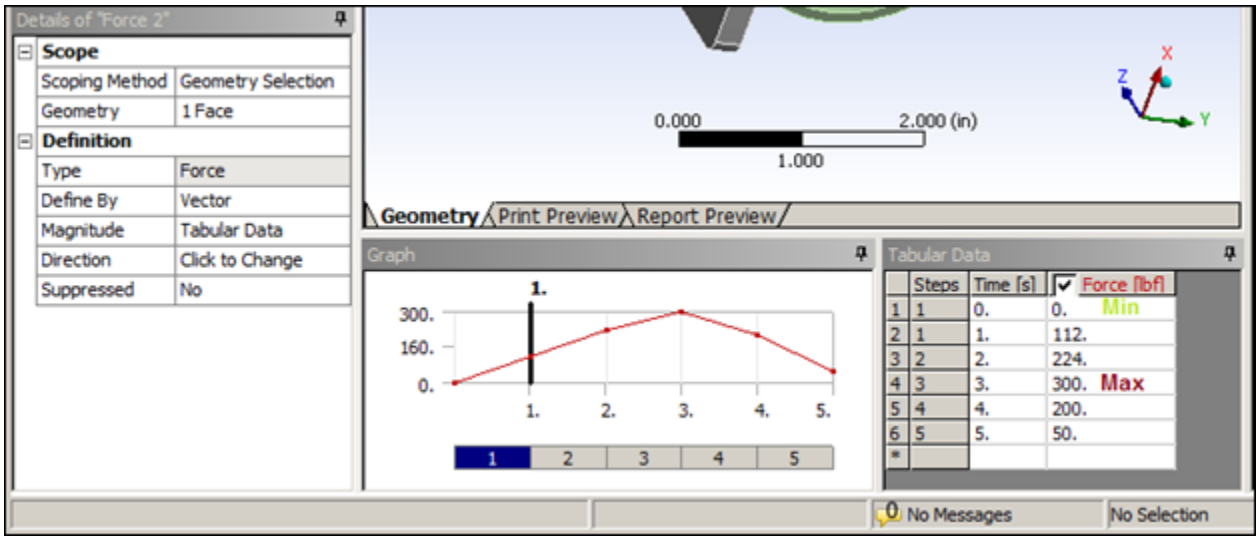
You can also change the force to be defined by components instead of by magnitude and set the values of an individual component:

```
force.DefineBy = LoadDefineBy.Components
force.ZComponent.Output.DiscreteValues = [Quantity('0 [N]'),Quantity('-9 [N]')]
```



Extracting Min-Max Tabular Data for a Boundary Condition

The Field API can be used to extract minimum and maximum tabular data for boundary conditions. For example, you can extract minimum and maximum applied force. The following figure displays tabular data for a force and highlights its minimum and maximum quantities.



First, get the project model object, analysis object, force object, and the output variable with the applied force variables:

```
mymodel = DataModel.Project.Model
anal = mymodel.Analyses[0]
f2 = anal.Children[2]
f2td = f2.Magnitude.Output
```

Next, get the tuple containing the minimum and maximum force quantities and then display these quantities by entering the variable name of the tuple:

```
mnmxf2 = f2td.MinMaxDiscreteValues
mnmxf2
```

Given the above tabular data, the following results display:

`(0 [lbf], 300 [lbf])`

The variable `mnxf2` is a tuple (pair) of quantities. Each element in the tuple can be gotten by using the tuple's properties **`Item1`** and **`Item2`**.

To get and display only the minimum force quantity:

```
mnv = mnmxf2.Item1
mnv
```

Given the above tabular data, the following results display:

`(0 [lbf])`

To get and display only the maximum force quantity:

```
mxv = mnmxf2.Item2
mxv
```

Given the above tabular data, the following results display:

`(300 [lbf])`

Setting the Direction of a Boundary Condition

You can set the direction of a boundary condition.

To set the direction of a load to a known value:

```
pressure.Direction = Vector3D(1,0,0)
```

To set the direction of a load using the direction from a face or edge or between points:

```
sels = ExtAPI.SelectionManager.CurrentSelection
reversed = True
vec = Ansys.Mechanical.Selection.SelectionHelper.CreateVector3D(sels, reversed)
acceleration.Direction = vec
```

Worksheets

You can use Mechanical APIs to access or modify data that is displayed by the user interface in the worksheet panel:

Named Selection Worksheet

Mesh Order Worksheet

Layered Section Worksheet

Bushing Joint Worksheet

Named Selection Worksheet

Mechanical APIs support all available actions for named selection worksheets, as described in Specifying Named Selections Using Worksheet Criteria in the *ANSYS Mechanical User's Guide*.

The following topics describe how to use APIs to define a named selection worksheet:

Defining the Named Selection Worksheet

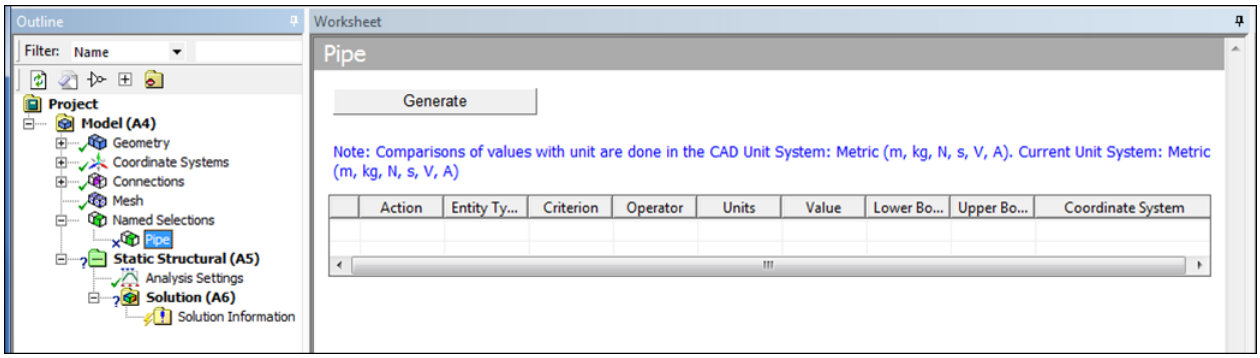
Adding New Criteria to the Named Selection Worksheet

Defining the Named Selection Worksheet

To define a named selection worksheet, you execute these commands:

```
sel = Model.AddNamedSelection()
sel.ScopingMethod=GeometryDefineByType.Worksheet
sel.Name = "Pipe"
pipews = sel.GenerationCriteria
```

This sample code creates the named selection object, renames it yo **`Pipe`**, changes the property **`Define By`** to **`Worksheet`**, and stores a variable of the NamedSelectionCriteria object that can be used to manipulate the data in the worksheet.



Adding New Criteria to the Named Selection Worksheet

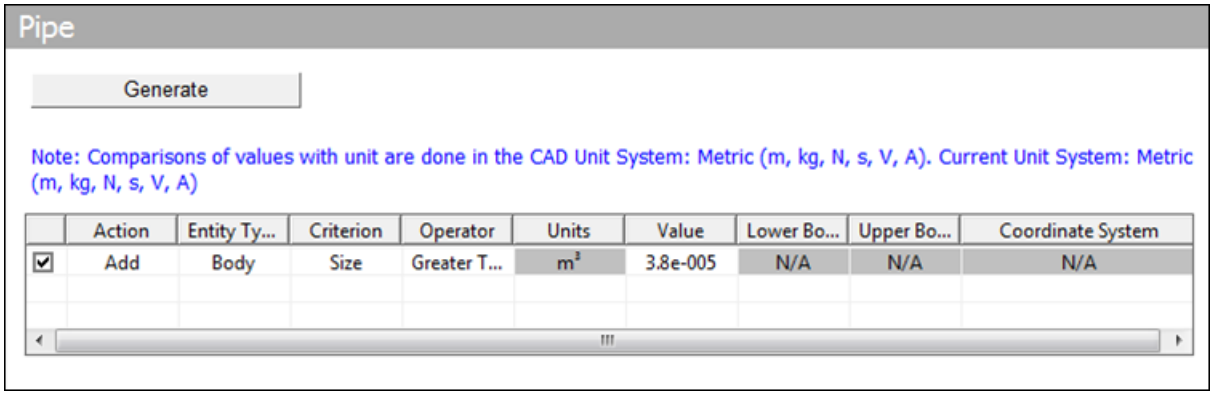
The named selection worksheet defines the criteria from which to generate a selection. Each row of the worksheet adds a new criteria and defines the relationship with the previous rows. The focus here is on adding a new row.

The **`NamedSelectionCriteria`** object behaves like a list and has standard list properties and methods including **`Add()`**, **`Clear()`**, **`Count`**, **`Insert`**, **`Remove`**, and a [] indexer. You can use the **`Add()`** method to add a NamedSelectionCriterion, which is the object for each element in the list and controls the data of a single row in the worksheet.

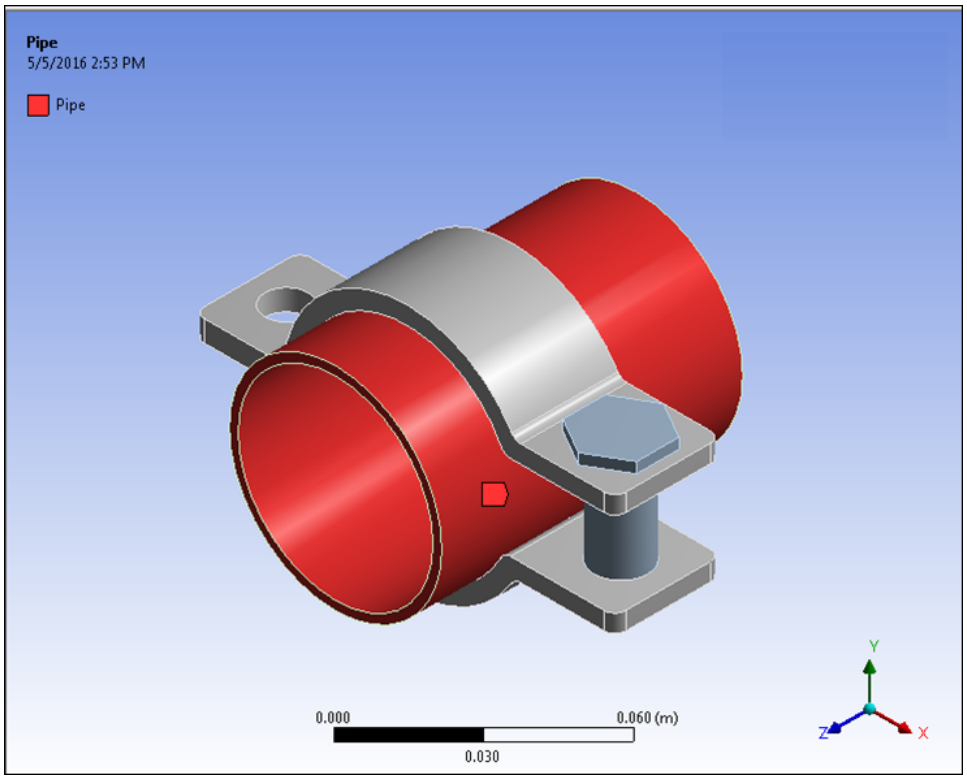
To add the first criterion to your worksheet:

```
pipews.Add(None)
pipews[0].EntityType=SelectionType.GeoBody
pipews[0].Criterion=SelectionCriterionType.Size
pipews[0].Operator=SelectionOperatorType.GreaterThan
pipews[0].Value=Quantity("3.8e-5 [m m m]")
sel.Generate()
```

This example defines a criterion to select bodies with a size value greater than **3.8e-5**.



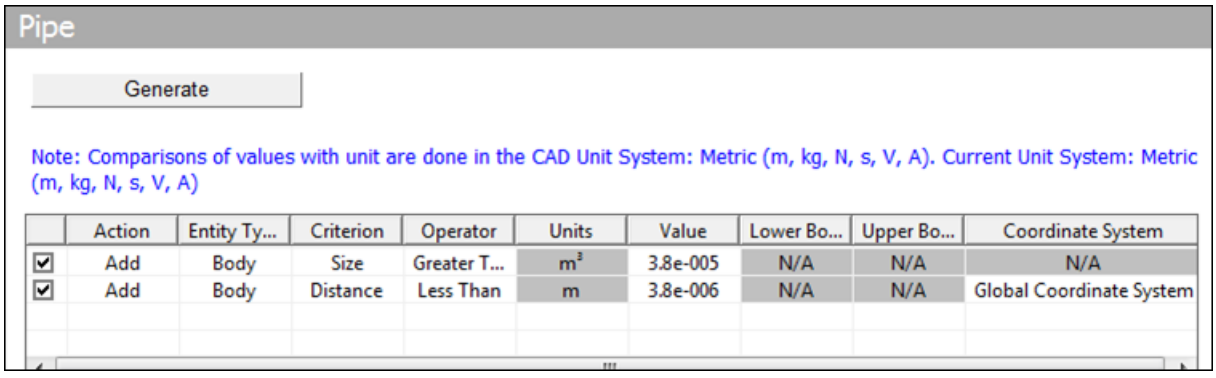
The method **`Generate`** on the **`NamedSelection`** object generates the selection based on the criteria. After executing this method, observe that the pipe body is selected in the **Graphics** view. In the **Details** view, the property **Total Selection** is set to **1 Body**.



Next, you can add another criteria to include other bodies:

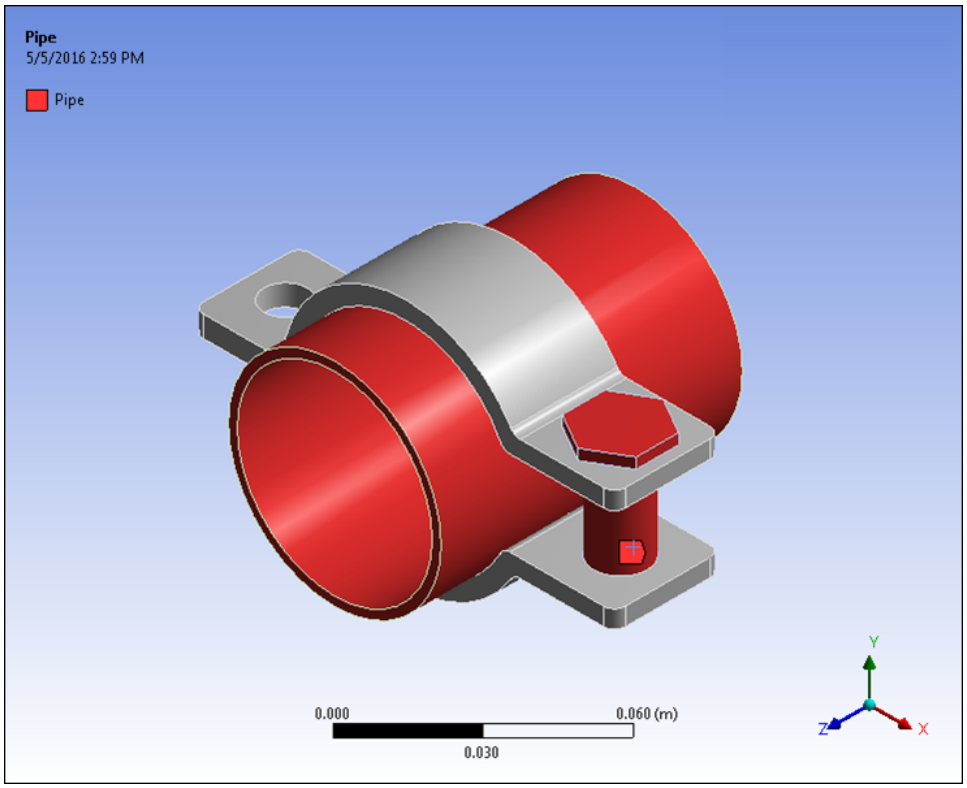
```
pipews.Add(None)
pipews[1].EntityType=SelectionType.GeoBody
pipews[1].Criterion=SelectionCriterionType.Distance
pipews[1].Operator=SelectionOperatorType.LessThan
pipews[1].Value=Quantity("3.8e-6 [m]")
sel.Generate()
```

This new criteria is defined to select bodies whose distance from a given coordinate system is less than **3.8e-6**. If you do not specify a coordinate system, it defaults to the global coordinate system.



	Action	Entity Ty...	Criterion	Operator	Units	Value	Lower Bo...	Upper Bo...	Coordinate System
☑	Add	Body	Size	Greater T...	m^3	3.8e-005	N/A	N/A	N/A
☑	Add	Body	Distance	Less Than	m	3.8e-006	N/A	N/A	Global Coordinate System

After executing the **`Generate`** method, observe that the bolt body is now selected in the **Graphics** view, along with the pipe body. In the **Details** view, the property **Total Selection** is set to **2 Bodies**.

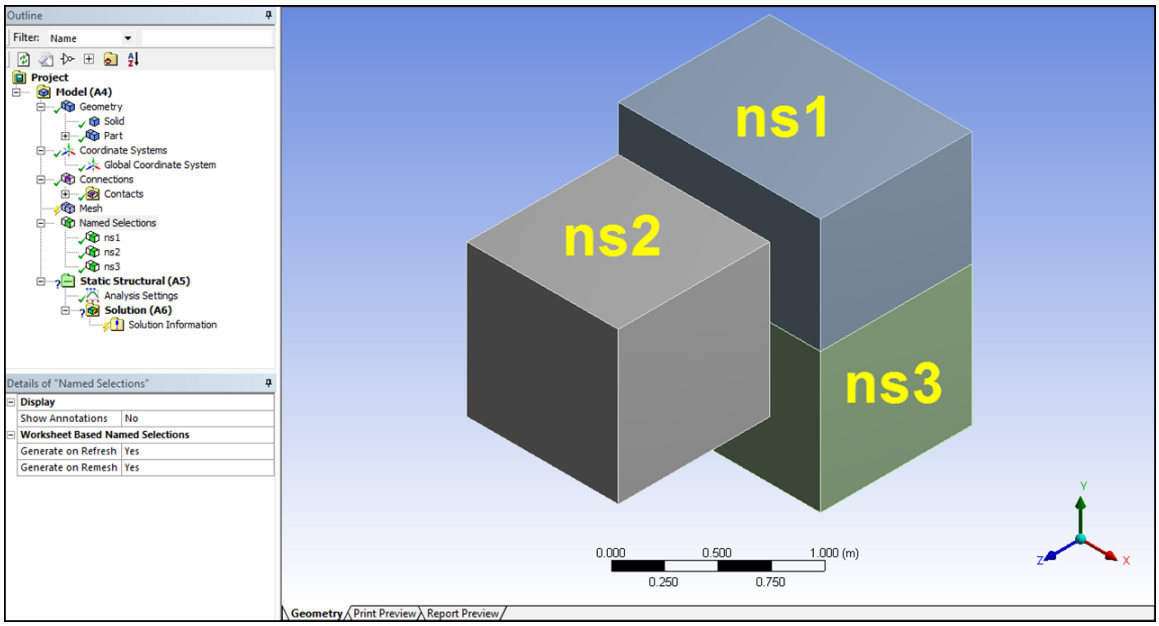


Mesh Order Worksheet

This section describes how to use Mechanical APIs to specify the order in which meshing steps are performed in Mechanical. This feature is described in Using the Mesh Worksheet to Create a Selective Meshing History in the *ANSYS Meshing User's Guide*. The following restrictions apply to these APIs:

- The named selections must have **Entity Type** set to **Body**. Other entity types are not supported.
- The **Start Recording** and **Stop Recording** buttons do not have APIs.

The examples in this section assume that you have already defined two or more named selections in Mechanical. In the following figure, three named selections are defined.



Defining the Mesh Worksheet

To begin using the APIs to define the Mesh Worksheet as described in the following example, use the following variables to refer to the mesh worksheet data and the three named selections to be used in the worksheet. The MeshControlWorksheet object is used to control the worksheet data.

```
mws = Model.Mesh.Worksheet
nsels = Model.NamedSelections
ns1 = nsels.Children[0]
ns2 = nsels.Children[1]
ns3 = nsels.Children[2]
```

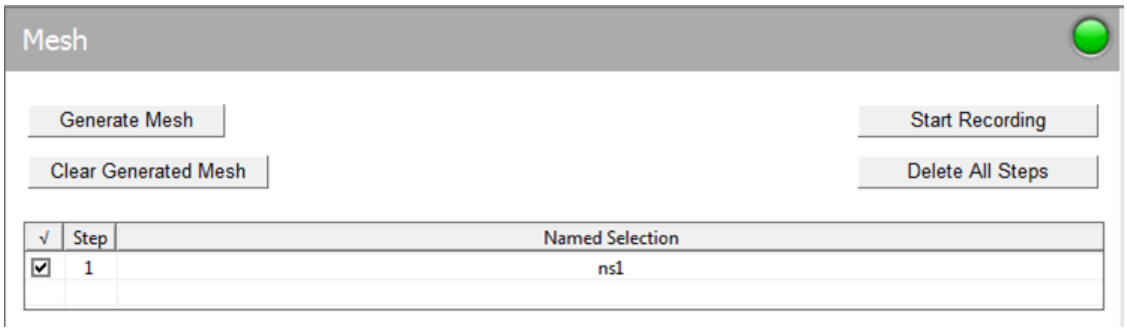
Adding New Rows in the Mesh Worksheet

To add the first row to your worksheet, you execute these commands:

```
mws.AddRow()
mws.SetNamedSelection(0,ns1)
mws.SetActiveState(0,True)
```

- The command **`AddRow`** adds an empty row.
- The command **`SetNamedSelection`** sets the value in the **Named Selection** column to your named selection variable **`ns1`**.
- The command **`SetActiveState`** sets **Active State** at the row index. This is indicated by the check mark in the left column.

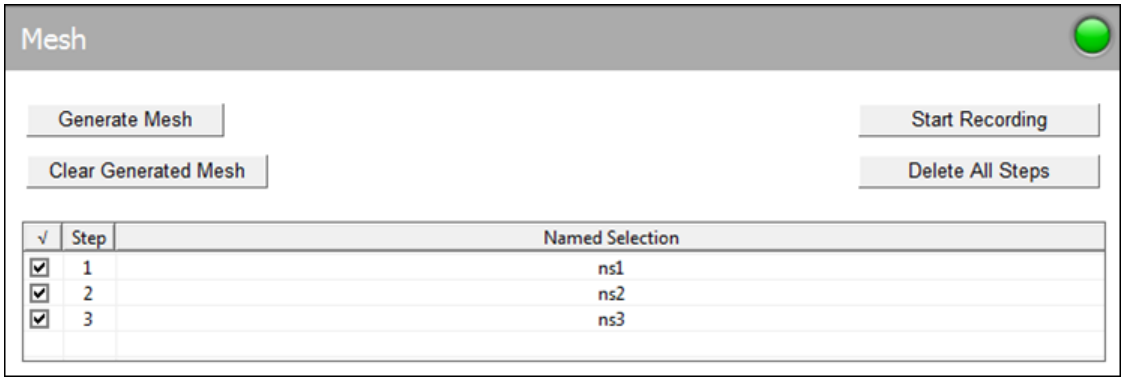
The following figure shows the mesh worksheet after the execution of these commands.



To add a row for each of the other two named selections, **ns2** and **ns3**, use these commands:

```
mws.AddRow()
mws.SetNamedSelection(1,ns2)
mws.SetActiveState(1,True)
mws.AddRow()
mws.SetNamedSelection(2,ns3.)
mws.SetActiveState(2,True)
```

The following figure shows the mesh worksheet after the two new rows have been added.

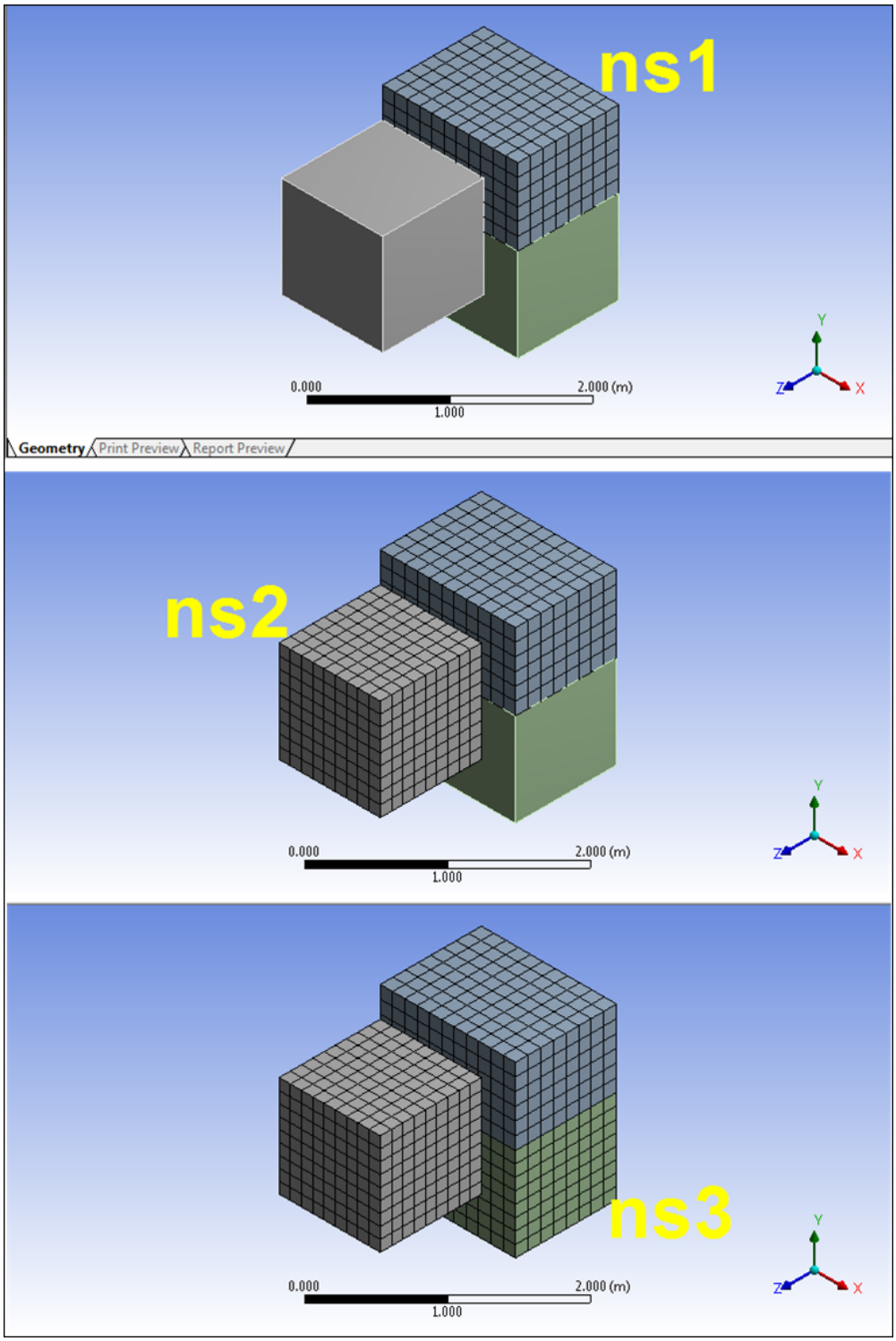


Meshing the Named Selections

When you check the **Graphics** tab, you can see that no mesh has been generated yet. To generate the mesh for the named selections in the mesh worksheet, execute the following command:

```
mws.GenerateMesh()
```

When you execute this command, the steps are processed one by one in the order specified by the worksheet. For each step, the bodies identified by the named selection are meshed using the meshing controls applied to them. By watching the mesh generation in the **Graphics** tab, you can see the order in which each body is meshed.



Layered Section Worksheet

To get to the LayeredSectionWorksheet object:

```
lsws = Model.Geometry.Children[1].Layers
```

This object uses 0-based indices to refer to rows.

- To control the data in the material column, use the **`GetMaterial`** or **`SetMaterial`** method.
- To control the data in the **`Thickness`** column, use the **`GetThickness`** or **`SetThickness`** method.
- To control the data in the **`Angle`** column, use the **`GetAngle`** or **`SetAngle`** method.

Bushing Joint Worksheet

A bushing joint worksheet defines stiffness and damping coefficients via symmetric matrices. The matrices are fixed size and symmetric fixed, which means that the API cannot add or remove entries or modify the upper right triangle. The following topics describe how to use the API to access a bushing joint worksheet:

Getting the Bushing Joint Worksheet and Its Properties

To begin using the APIs to define the bushing joint worksheet as described in the following example, use the following variable to refer to the bushing joint worksheet data. It will be an instance of the JointBushingWorksheet object.

```
bws = Model.Connections.Children[1].Children[0].BushingWorksheet
```

Getting the Value for a Given Component

The API uses 0-based row indices using a method for each column. Because only the lower left triangle can be controlled, each method will have a different range of valid indices.

For example, to get the coefficient for the stiffness per unit Y at row index 1:

```
bws.GetBushingStiffnessPerUnitY(1)
```

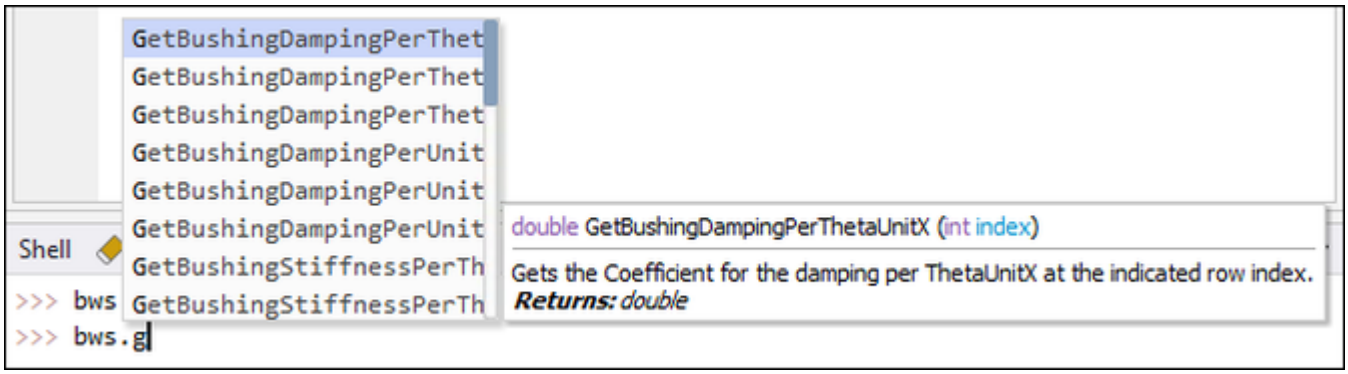
Assume that **`136073.550978543`** displays as the stiffness per unit Y.

Now, assume you have entered the following:

```
bws.GetBushingStiffnessPerUnitZ(0)
```

The given index of 0 (zero) produces an error message because the cell indicated is in the upper right triangle. As per the error message, the valid range of indices is 2 to 5, inclusive.

The following figure shows some of the many methods available for getting coefficients for damping and stiffness using the Mechanical **Scripting** view's autocomplete feature.


GetBushingDampingPerThet
GetBushingDampingPerThet
GetBushingDampingPerThet
GetBushingDampingPerUnit
GetBushingDampingPerUnit
GetBushingDampingPerUnit
GetBushingStiffnessPerTh
GetBushingStiffnessPerTh
double GetBushingDampingPerThetaUnitX (int index)
Gets the Coefficient for the damping per ThetaUnitX at the indicated row index.
Returns: double
Shell
>>> bws
>>> bws.g

Graphics

This section describes APIs related to graphics:

Manipulating Graphics

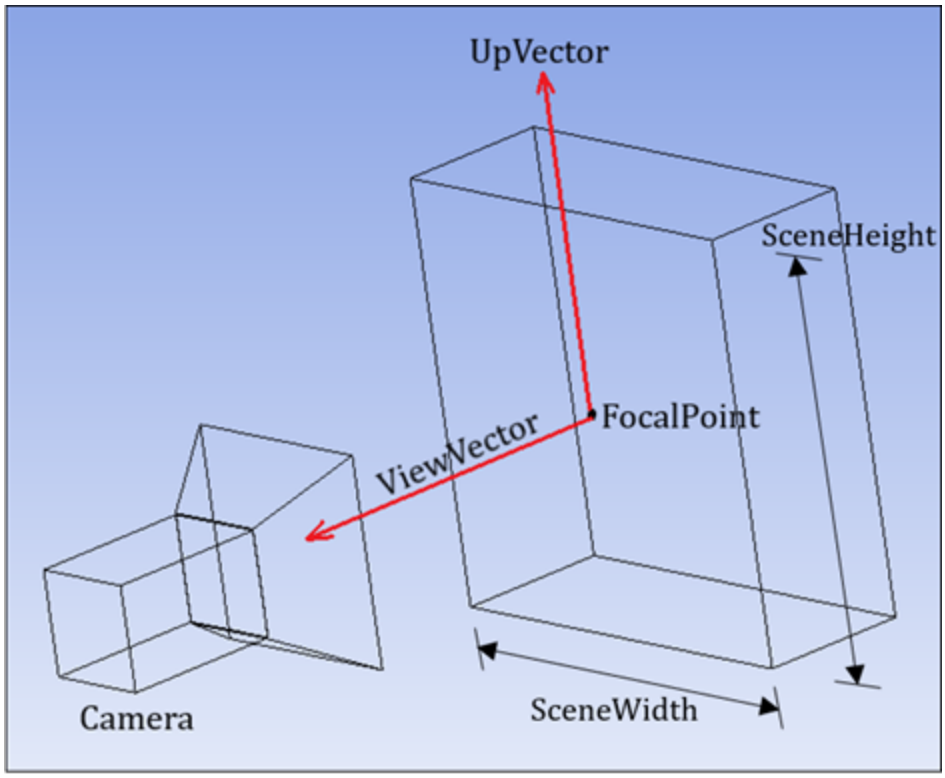
You can use the MechanicalCameraWrapper to manipulate the camera for precise control of model visualization.

To access this API, you enter these commands in the console:

```
camera = Graphics.Camera
camera
```

The following properties together represent the state of the camera view:

Property	Description
`FocalPoint`	Gets or sets the focal point of the camera. Coordinates are in the global coordinate system
`SceneHeight`	Gets or sets the scene height (in length units) that will be projected and fit to the viewport
`SceneWidth`	Gets or sets the scene width (in length units) that will be projected and fit to the viewport
`UpVector`	Gets or sets the vector pointing up from the focal point
`ViewVector`	Gets or sets the vector pointing from the focal point to the camera



An example follows for using these properties to change the camera state:

```
camera.FocalPoint = Point((0.0,0.0,0.0), "mm")
camera.ViewVector = Vector3D(1.0,0.0,0.0)
camera.UpVector = Vector3D(0.0,1.0,0.0)
camera.SceneHeight = Quantity(100, “mm”)
camera.SceneWidth = Quantity(150, “mm”)
```

In addition to changing one or more specific properties, user-friendly methods are available for manipulating the camera state.

- To fit the view to the whole model or selection:

  ```
  camera.SetFit(ISelectionInfo selection = null)
  ```

- To rotate along a particular axis (global or screen):

  ```
  camera.Rotate(double angle, CameraAxisType axisType)
  ```

- To set a specific view orientation:

  ```
  camera.SetSpecificViewOrientation(ViewOrientationType orientationType)
  ```

Two examples follow.

Example 1

This code rotates the model by 30 degrees about the direction normal to the current screen display:

```
camera.Rotate(30, CameraAxisType.ScreenZ)
```

Example 2

These code samples fit the view to a selection (either a face with a known ID or a named selection).

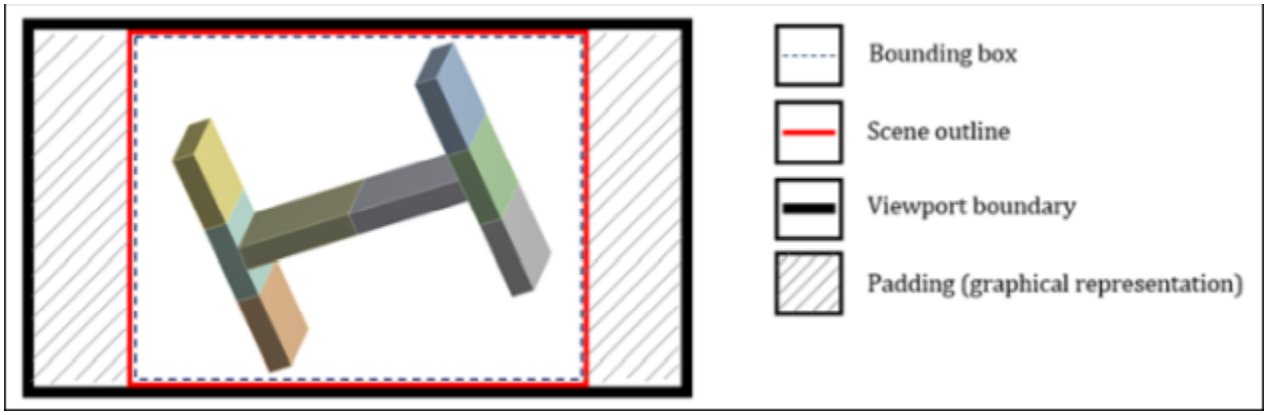
```
#fit view to face #28
selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
selection.Ids = [28]
camera.SetFit(selection)


#fit view to the first named selection
named_selection = Model.NamedSelections.Children[0]
camera.SetFit(named_selection)
```

SceneHeight and SceneWidth Usage

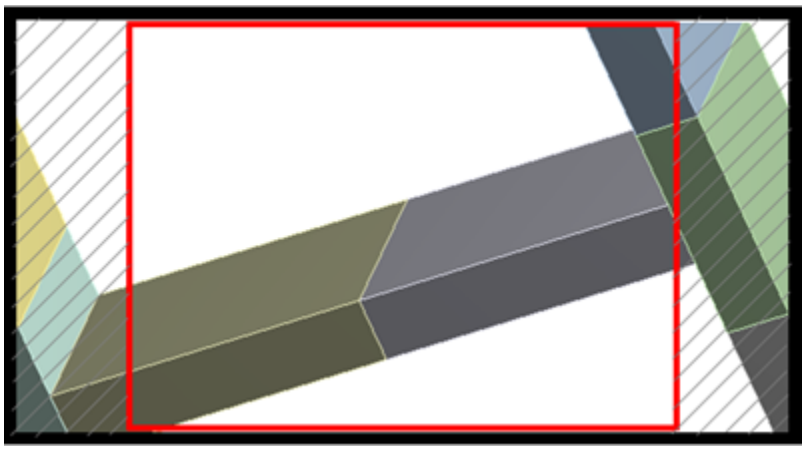
The two properties **`SceneHeight`** and **`SceneWidth`**, along with the existing camera APIs (such as **`FocalPoint`**, **`UpVector`**, and **`ViewVector`**) determine the volume of the scene. This volume defines the minimum view that will be visible (without any distortion) after projection onto a 2D viewport in graphics (which may have a different aspect ratio.) These quantities are in length units and can only be affected by zoom operations like zoom in/out, zoom to fit, and box zoom. This means that for a graphics window of a given aspect ratio, the extent defined by these two properties will always be seen on the screen (sometimes with additional "padding" depending on the aspect ratio).

An example scenario follows:

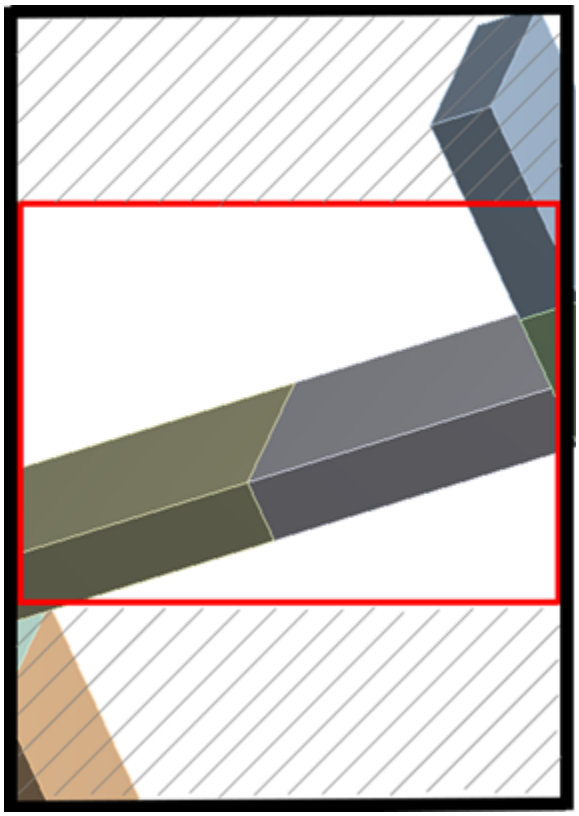
1. Open a model and zoom to fit. The **`SceneHeight`** and **`SceneWidth`** are equal to the bounding box of the directed view. There is horizontal padding because the viewport boundary is larger than the bounding box in that dimension.



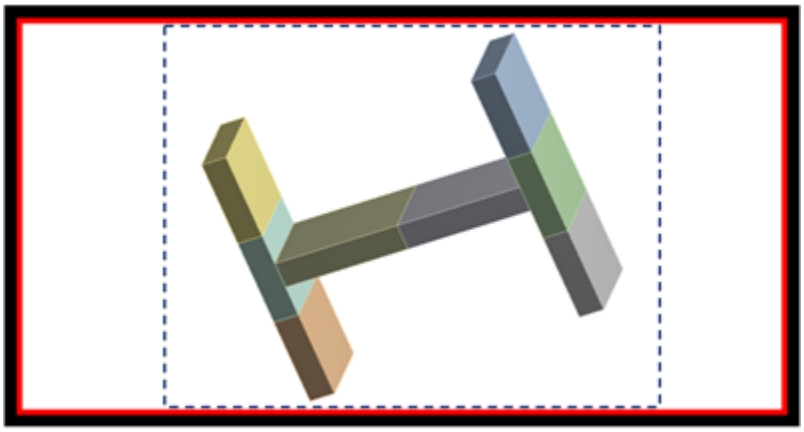
2. If you set **`SceneHeight`** and **`SceneWidth`** to one-third ( ) of their original values, the camera zooms in. The deciding factor is **`SceneHeight`** because of the screen's aspect ratio and orientation. The scene, defined by **`SceneWidth`** and **`SceneHeight`**, is then scaled in or out, maintaining the aspect ratio until the height equals the viewport height.



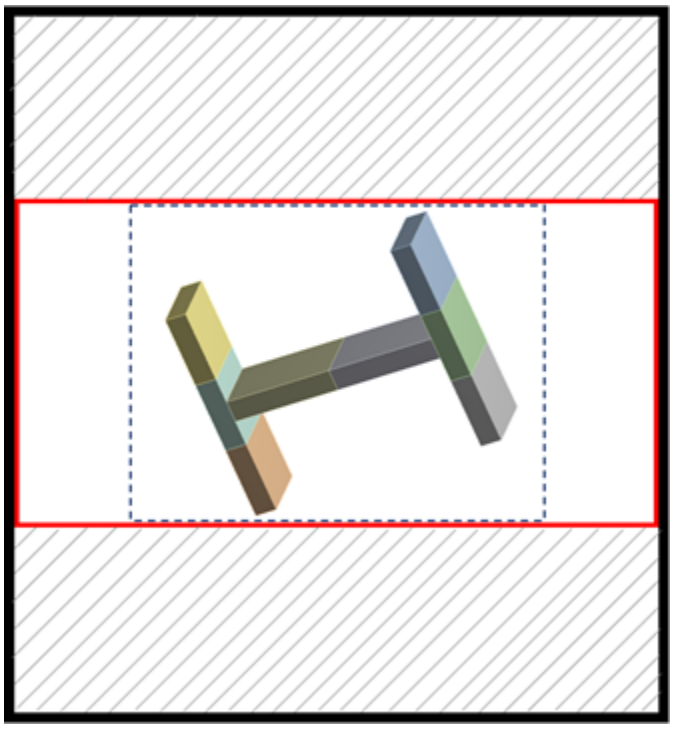
3. If you now resize the viewport panel to be taller and more slender, the scene defined by **`SceneHeight`** and **`SceneWidth`** is then scaled in or out, maintaining the aspect ratio until the width equals the viewport width.

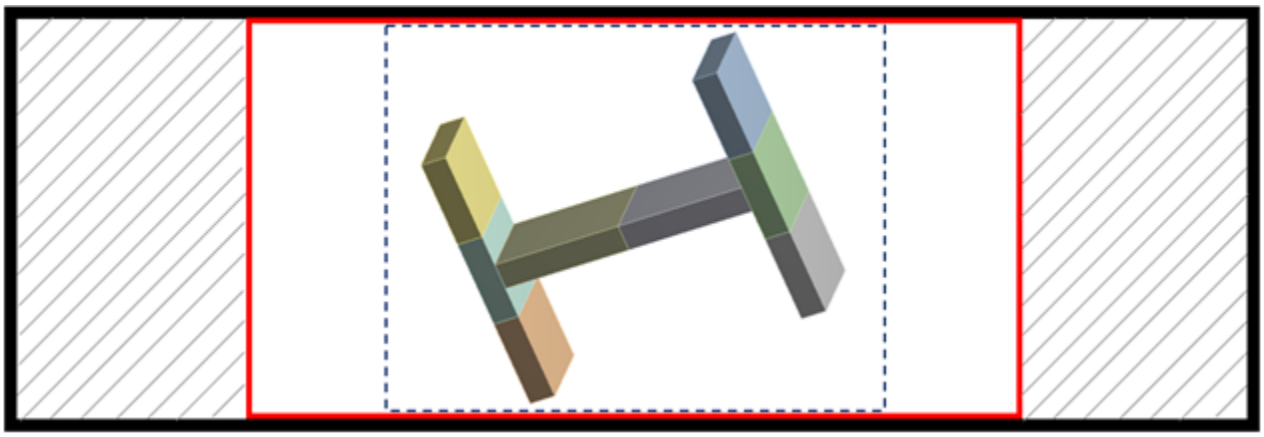


4. Let's say after point 1, you increase the **`SceneWidth`** to something beyond the model bounding box (such that it includes the padding space).



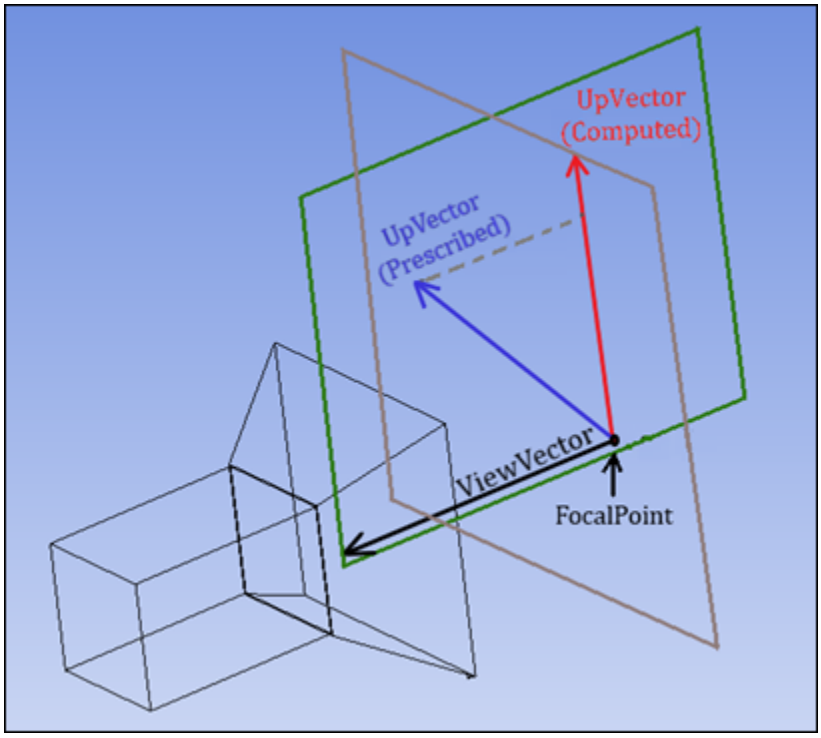
5. If you enlarge the viewport panel in either direction, the scene parameters still hold their shape.





UpVector Usage

Mathematically, the **UpVector** and **ViewVector** of the camera must be perpendicular. Together they define the camera orientation, which in the following figure is shown by the green half-plane. However, you can use the API to prescribe any **UpVector** that is not collinear with the **ViewVector**. The camera orientation's **UpVector** is then internally computed to use the projection (dashed line) of the prescribed **UpVector** (blue line) onto the brown plane perpendicular to the **ViewVector** (black line). The camera still reports back the prescribed **UpVector** with the **UpVector** property.



Note:

In the half-plane, there are an infinite number of prescribed **UpVector** choices that result in the same computed **UpVector**.

Exporting Graphics

You can use the methods **Export3D** and **ExportImage** to export graphics to 3D model files (STL and AVZ) and image files (PNG, JPG, TIF, BMP, and EPS).

To export graphics to model and image files, you use the following commands:

Command	Description
Graphics.Export3D	Exports the model to an STL or AVZ file
Graphics.ExportImage	Exports the image to a PNG, JPG, TIF, BMP, or EPS file

This code exports the model to a binary STL file:

```
setting3d = Ansys.Mechanical.Graphics.Graphics3DExportSettings()
Graphics.Export3D("c:\\temp\\binarySTL.stl", Graphics3DExportFormat.BinarySTL, setting3d)
```

This code exports the image to a PNG file:

```
setting2d = Ansys.Mechanical.Graphics.GraphicsImageExportSettings()
Graphics.ExportImage("c:\\temp\\imagePNG.png", GraphicsImageExportFormat.PNG, setting2d)
```

For information on changing the 3D model or image export settings, see Graphics3DExportSettings or GraphicsImageExportSettings in the *ACT API Reference Guide*.

The following script exports the model with a white background to an AVZ file and exports the image with enhanced resolution to a PNG file.

```
#export model to AVZ file with white background
setting3d = Ansys.Mechanical.Graphics.Graphics3DExportSettings()
setting3d.Background = GraphicsBackgroundType.White
Graphics.Export3D("c:\\avz_white.avz", Graphics3DExportFormat.AVZ, setting3d)

#export image with enhanced resolution to PNG file
setting2d = Ansys.Mechanical.Graphics.GraphicsImageExportSettings()
setting2d.Resolution = GraphicsResolutionType.EnhancedResolution
Graphics.ExportImage("c:\\temp\\image_enhancement.png", GraphicsImageExportFormat.PNG, setting2d)
```

Exporting Result or Probe Animations

You can use the method **`ExportAnimation`** to export a result or probe animation to a video format (MP4, WMV, AVI, or GIF).

When exporting an animation to a video file, you can set the file name, format, and export settings.

- **`GraphicsAnimationExportFormat`** sets the file format to which to save the animation.
- **`AnimationExportSettings`** sets the width and height properties for the file. When this is not set, the resolution for the current graphics display is used to set the width and height.

Example 1

This code exports a result animation to a MP4 video file:

```
#Exporting a result animation to mp4 video file
totalDeform = DataModel.GetObjectsByName("Total Deformation")[0]
totalDeform.ExportAnimation("E:\\file.mp4",GraphicsAnimationExportFormat.MP4)
```

Example 2

This code exports a result animation to a WMV file with a specific resolution (width=1000,height=665):

```
#Exporting a result animation to wmv video file with given size
totalDeform1 = DataModel.GetObjectsByName("Total Deformation 1")[0]
settings = Ansys.Mechanical.Graphics.AnimationExportSettings(width = 1000, height = 665)
totalDeform1.ExportAnimation("E:\\file.wmv",GraphicsAnimationExportFormat.WMV,settings)
```

Animation Settings

To control the behavior of the animation for all results, you can specify global animation settings related to the toolbar visible when the Animation feature is used:

Property	Description
Graphics.ResultAnimationOptions.NumberOfFrames	Gets or sets the number of frames for the distributed result animation range type
Graphics.ResultAnimationOptions.Duration	Gets or sets the range type of the result animation
Graphics.ResultAnimationOptions.TimeDecayCycles	Gets or sets the number of cycles for the time decay analysis
Graphics.ResultAnimationOptions.UpdateContourRangeAtEachFrame	Gets or sets if the legend contours update at each frame
Graphics.ResultAnimationOptions.FitDeformationScalingToAnimation	Gets or sets the fit deformation scaling at each range for the full range for a result that has multiple time steps

This code sets the number of frames to 200:

```
#set the number of frames to 200
Graphics.ResultAnimationOptions.NumberOfFrames = 200
```

This code sets the play duration to 12 seconds:

```
#set play duration to 12 seconds
Graphics.ResultAnimationOptions.Duration = Quantity(12, "s")
```

Creating Section Planes

You can use the SectionPlanes object, which is a collection of individual section planes, to add a section plane on your model to view the interior of the geometry, mesh, or results or to view the shape of the cross section. The **SectionPlanes** collection can have any number of section planes, but no more than six section planes can be active at once.

To access the collection associated with the Section Planes window, you use this command:

```
Graphics.SectionPlanes
```

The items in the collection use the **SectionPlane** object. The following methods and properties can be used to manipulate the collection:

Command	Description
SectionPlanes.Add(SectionPlane)	Add a new section plane to the collection
SectionPlanes.Remove(SectionPlane)	Remove a new section plane from the collection
SectionPlanes.RemoveAt(0)	Remove a section plane at a given index
SectionPlanes.Clear()	Clear all the section planes from the collection

Command	Description
`section_plane = SectionPlanes[0]`	Get a section plane at an index

Along with methods and properties to manipulate the collection, the **`SectionPlanes`** object has two global properties that apply to all section planes. These global properties are shown in the **Graphics** window:

Command	Description
`SectionPlanes.Capping`	Capping style of the section plane
`SectionPlanes.ShowWholeElement`	Element visibility of the section plane

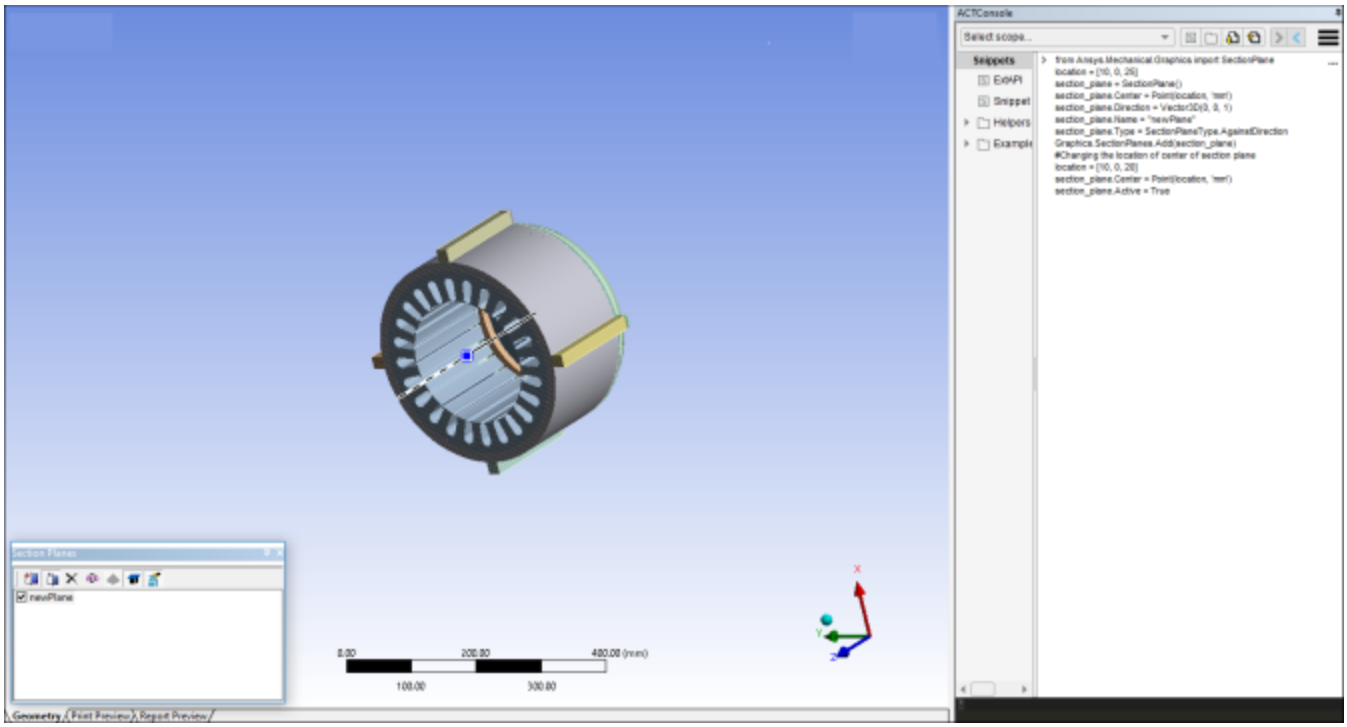
Each individual section plane in the collection contains these properties:

Command	Description
`Graphics.SectionPlanes[0].Center`	Center point of the section plane
`Graphics.SectionPlanes[0].Type`	Type of the section plane
`Graphics.SectionPlanes[0].Name`	Name of the section plane
`Graphics.SectionPlanes[0].Active`	Active state of the section plane
`Graphics.SectionPlanes[0].Direction`	Normal direction of the section plane

Example 1

This code creates a section plane and then both adds and changes the location:

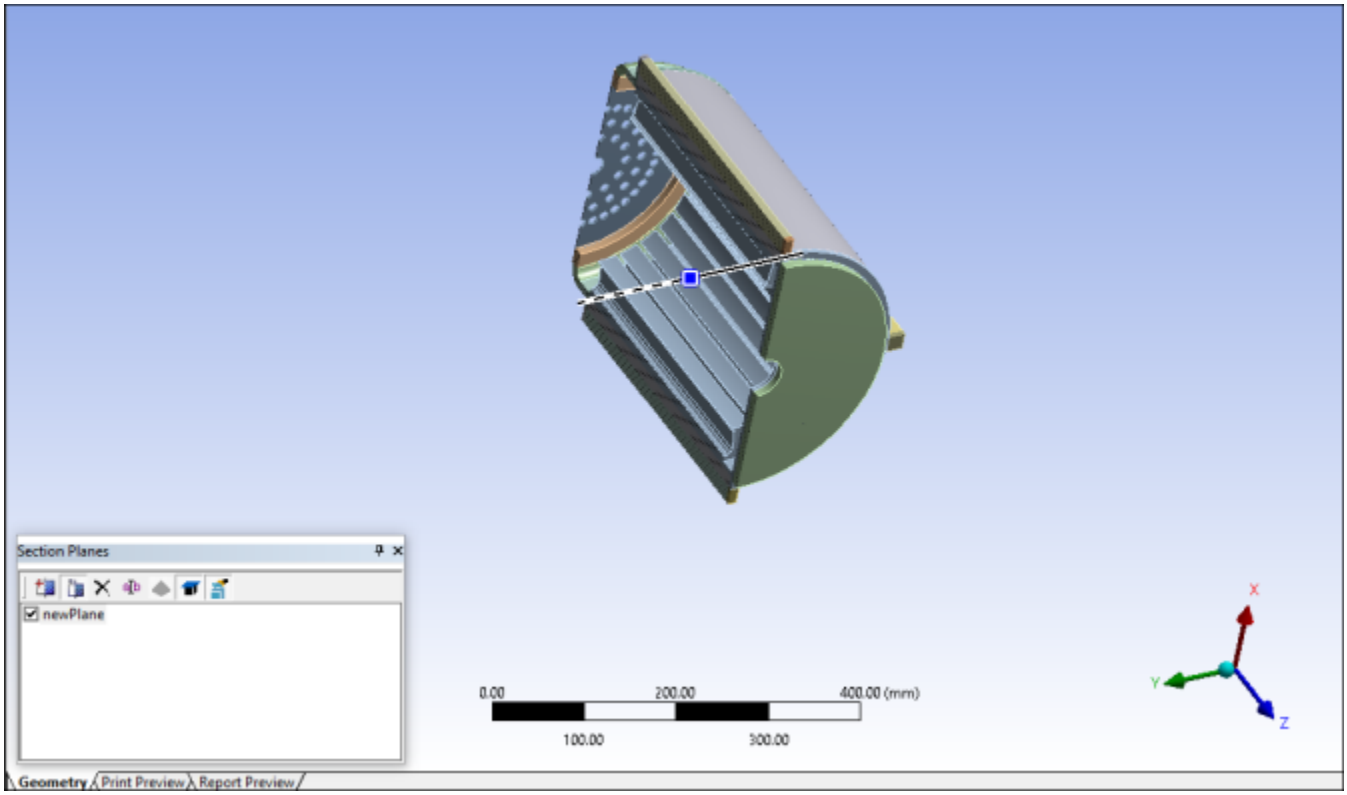
```
from Ansys.Mechanical.Graphics import SectionPlane
location = [100, 150, 255]
sectionPlane = SectionPlane()
sectionPlane.Center = Point(location, 'mm')
sectionPlane.Direction = Vector3D(0, 0, 1)
sectionPlane.Name = "newPlane"
sectionPlane.Type = SectionPlaneType.AgainstDirection
Graphics.SectionPlanes.Add(sectionPlane)
location = [100, 150, 200]
sectionPlane.Center = Point(location, 'mm')
```



Example 2

This code modifies the direction of the section plane:

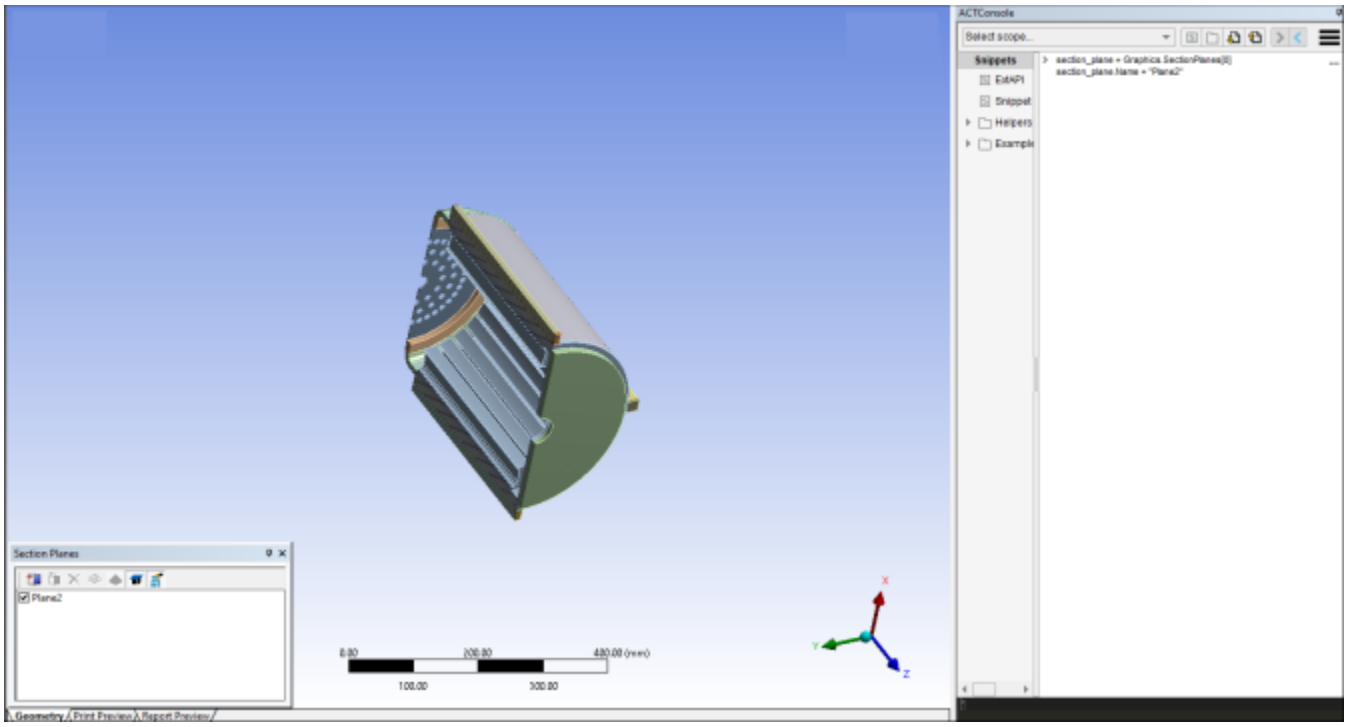
```
sectionPlane.Direction = Vector3D(0, 1, 0)
```



Example 3

This code gets the existing section plane and changes its name:

```
sectionPlane = Graphics.SectionPlanes[1]
sectionPlane.Name = "Plane2"
```



Example 4

As indicated earlier, while a **`SectionPlanes`** collection can have any number of **`SectionPlane`** objects, no more than six **`SectionPlane`** objects can be active at once. Assume that your collection holds six **`SectionPlane`** objects, which are all in an active state. If you try to add a new **`SectionPlane`** object with the property **`Active`** set to **`True`**, the **`SectionPlane`** object is added to the collection. However, the property **`Active`** is automatically set to **`False`** and an error message is shown:

```
> ExtAPI.Graphics.SectionPlanes.Count
< 6
> from Ansys.Mechanical.Graphics import SectionPlane
  location = [100, 150, 255]
  sectionPlane = SectionPlane()
  sectionPlane.Center = Point(location, 'mm')
  sectionPlane.Direction = Vector3D(0, 0, 1)
  sectionPlane.Name = "newPlane"
  sectionPlane.Type = SectionPlaneType.AgainstDirection
  sectionPlane.Active = True
  Graphics.SectionPlanes.Add(sectionPlane)
⊗ Activation of the clip plane is not successful as the limit of 6 active clip planes has been reached.
> ExtAPI.Graphics.SectionPlanes.Count
< 7
> sectionPlane.Active
< False
```

While the collection now holds seven **`SectionPlane`** objects, only 6 are active. If you tried to activate the seventh **`SectionPlane`** object by setting its property **`Active`** to **`True`**, the property is automatically set back to **`False`** and an error message again displays that the limit of 6 active planes has been reached:

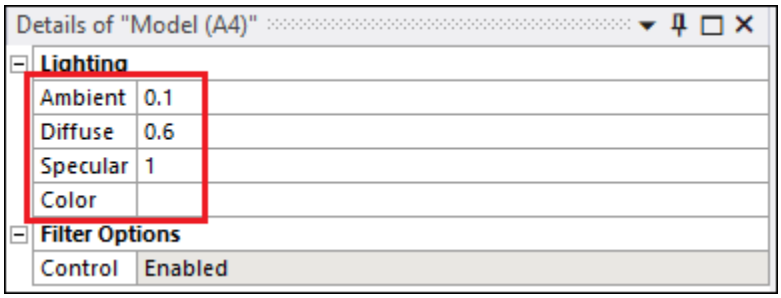
```
> ExtAPI.Graphics.SectionPlanes[6].Active
< False
> ExtAPI.Graphics.SectionPlanes[6].Active=True
⊗ Activation of the clip plane is not successful as the limit of 6 active clip planes has been reached.
> ExtAPI.Graphics.SectionPlanes[6].Active
< False
```

Setting Model Lighting Properties

You use the following properties to set model lighting:

Property	Description
`Ambient`	Gets or sets the ambient lighting factor
`Diffuse`	Gets or sets the diffuse lighting factor
`Specular`	Gets or sets the specular lighting factor
`Color`	Gets or sets the lighting color

The following image shows model lighting properties being set in the Details view for the model:



Example 1

This code sets model lighting properties:

```
Model.Ambient = 0.5
Model.Diffuse = 0.3
Model.Specular = 0.2
Model.Color = 13796830
```

Example 2

When setting model lighting color, you can also use hex color codes, which are three-byte hexadecimal numbers consisting of the prefix **`0x`** followed by six digits. Each byte (or pair of characters) following the prefix represents the intensity of red, green, and blue, respectively. For example, the hex color code for white is **`0xFFFFFF`**.

This code shows how to use a hex color code to specify a lilac shade ( ) for the model lighting color:

```
Model.Color = 0xD285DE
```

Results

In Mechanical, the **Solution** object contains results. To access the **Solution** object:

```
solution = Model.Analyses[0].Solution
```

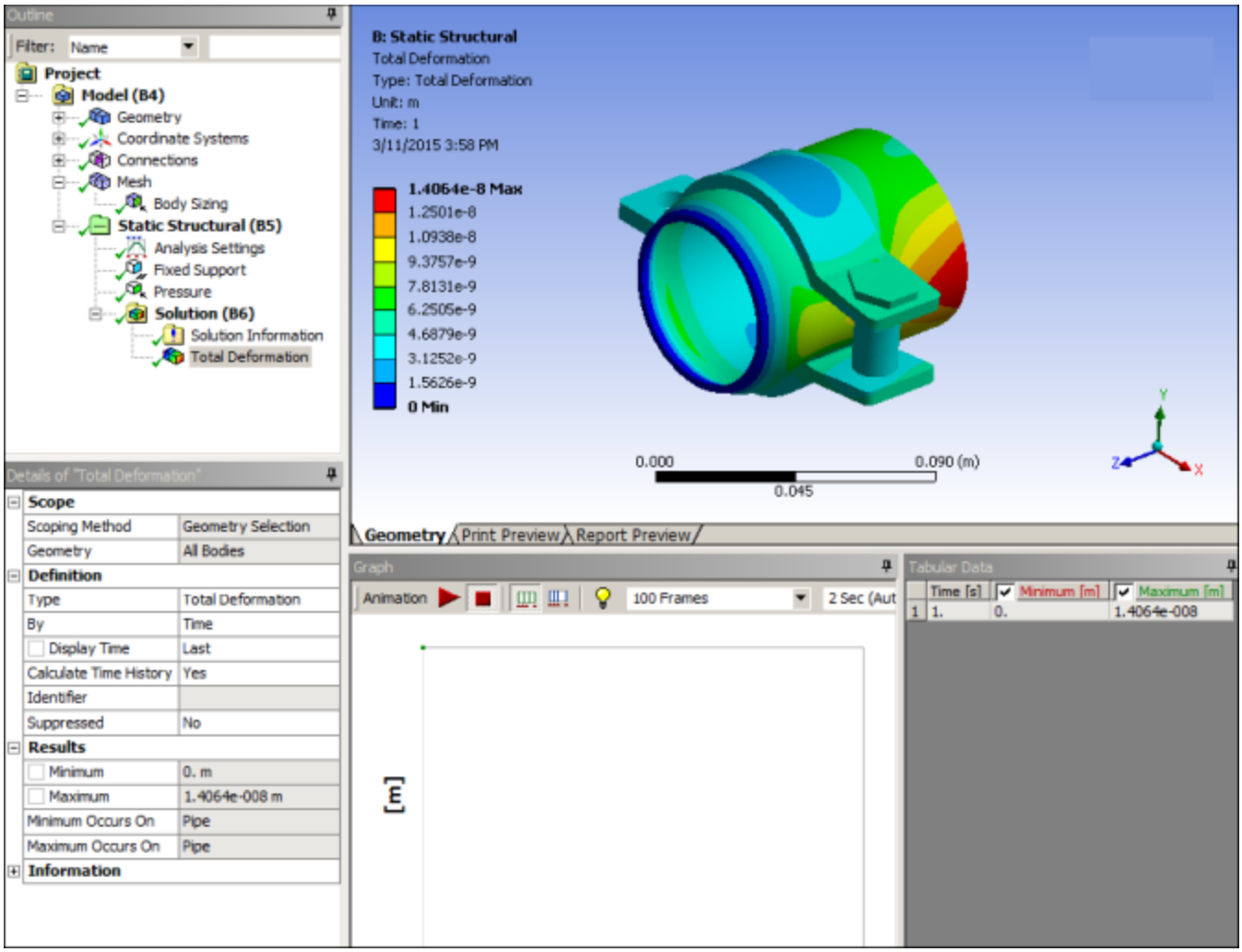
The following topics describe ways of using the API to add and work with results:

Adding Results to a Solution Object

You can use the API to add results to the **Solution** object. For example, you can add the **Total Deformation** result to a static structural analysis and then retrieve the minimum and maximum total deformation:

```
total_deformation = solution.AddTotalDeformation()
analysis = Model.Analyses=[0]
analysis.Solve(True)
minimum_deformation = total_deformation.Minimum
maximum_deformation = total_deformation.Maximum
```

It results in a solved analysis indicating the values for the properties **Minimum** and **Maximum** for the result **Total Deformation**.



Accessing Contour Results for an Evaluated Result

You can access contour results for an evaluated result in the tabular data interface. The result is represented in a table with independent and dependent column variables.

- The node and element IDs are independent variables (unique identifiers) of the tabular result.
- The result value components are dependent variables (based on the independent variables).

Contour results can be of three types:

- **Nodal**: Results are calculated on each node (like displacements)
- **Elemental**: Results are calculated on each element (like volumes)
- **ElementalNodal**: Results are calculated on the element but then interpolated onto the nodes (like stresses)

Contour results can have various styles:

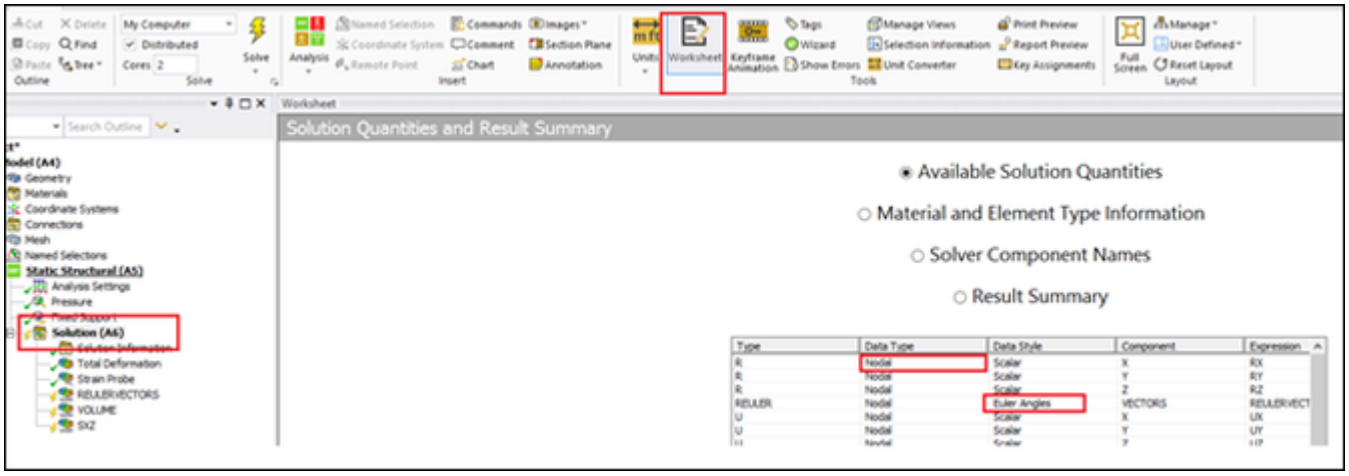
- **Scalar**: Has single result component
- **Vector**: Has X, Y, and Z components

- **Tensor**: Has X, Y, Z, XY, YZ, and XZ components
- **TensorStrain**: Has X, Y, Z, XY, YZ, and XZ components
- **EulerAngels**: Has XY, YZ, and XZ components
- **Coordinate**: Has X, Y, and Z components
- **ShearMomentDiagram**: Has MY,MZ,SFY,SFZ,UY, UZ, MSUM, SFSUM, and USUM components

Creating Contour Results of a Specific Type and Style

In Mechanical, you can create a contour result of a specific type and style and evaluate it. Using the **ACT Console**, you can then display contour results in the tabular data interface:

1. In the **Outline** view, select **Solution**, which activates the **Worksheet** tab.
2. Click the **Worksheet** tab. Results of various types display.



3. Right-click a result in the table and select **Create User Defined Result**.
4. In the **Outline** view, right-click and select **Evaluate All Results** to generate contour results for the new user-defined result.
5. To display the contour results in the tabular data interface, enter these commands:

```
result=Model.Analyses[0].Solution.Children[1]
result.PlotData
```

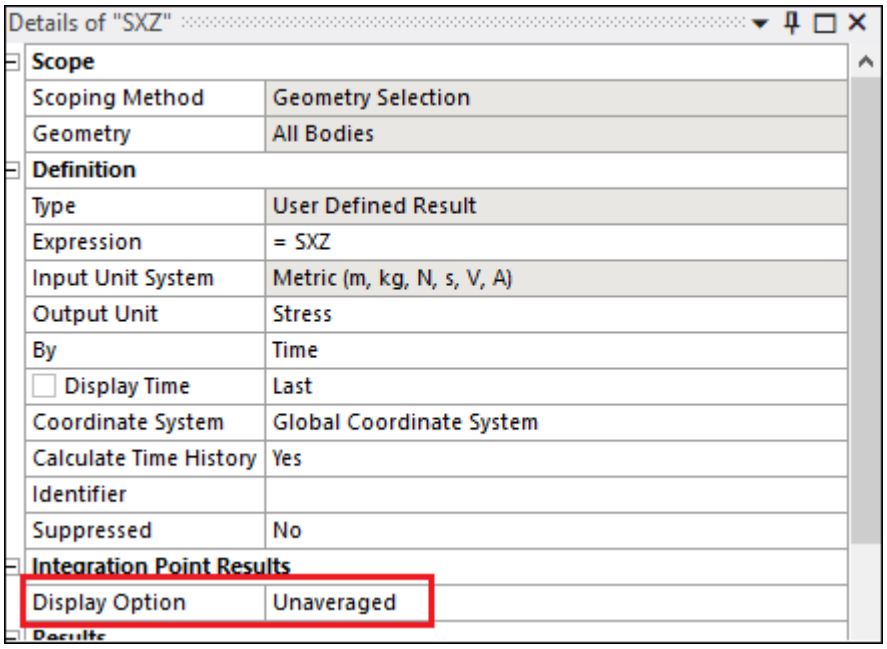
For a **Nodal** type, the results table has only node information:

```
>>> nodal = Model.Analyses[0].Solution.Children[5]
    nodal.PlotData
          Node            X            Y            Z
                        (m)          (m)          (m)
0            1  -9.008E-09  -7.2911E-10  1.3684E-10
1            2  -8.4171E-09  -7.2911E-10   1.443E-10
2            3  -7.8262E-09   -7.291E-10  1.5172E-10
...
3708      3709            0  -2.0808E-09           0
3709      3710            0  -1.4863E-09           0
3710      3711            0  -8.9179E-10           0
```

For an **Elemental** type, the results table has only element information:

```
>>> elemental = Model.Analyses[0].Solution.Children[6]
    elemental.PlotData
       Element           XY           YZ           XZ
                       (rad)        (rad)        (rad)
0            1            0            0            0
1            2            0            0            0
2            3            0            0            0
...
573        574       1.5708            0            0
574        575       1.5708            0            0
575        576       1.5708            0            0
```

For an **ElementalNodal** type, you must select an **ElementalNodal** result from the worksheet and evaluate with **Unaveraged** set for **Display Option** in the **Details** view of the result:



When you display contour results for an **ElementalNodal** type in the tabular data interface, the results table has both node and element information:

```
>>> elemnodeUA = Model.Analyses[0].Solution.Children[9]
    elemnodeUA.PlotData
             Node      Element         EPEL1
                                     (m)/(m)
0               1            1   5.4506E-13
1               2           13   1.1473E-12
2               3           25   1.6998E-12
...
11517        3705          568   1.2076E-08
11518        3706          569   1.0092E-08
11519        3707          567   1.2616E-08
```

Similarly, you can generate contour results with different result styles. Here is the command and output for an unaveraged tensor strain:

```
TabularDataInt... : Description
1  # Elemental Nodal Un Averaged
2  print(" Result 8 : Element Nodal Un Averaged")
3  elemnodeUA = Model.Analyses[0].Solution.Children[8]
4  elemnodeUA.PlotData
5
```

```
Shell
                                    Script Executed
 Result 8 : Element Nodal Un Averaged
             Node      Element           X   ...          XY          YZ          XZ
                                     (m)/(m)   ...     (m)/(m)     (m)/(m)     (m)/(m)
0               1            1   7.3737E-14   ... -7.9751E-13   9.7679E-13   4.4396E-13
1               2           13  -2.1989E-14   ... -1.1997E-12   1.9175E-12   2.1251E-13
2               3           25  -1.7415E-13   ... -7.4623E-13   3.2005E-12   1.2226E-13
...
11517        3705          568  -1.1832E-10   ... -2.8881E-08   1.1076E-10  -8.6136E-11
11518        3706          569  -1.7703E-10   ... -2.6536E-08   3.2731E-09   -3.479E-11
11519        3707          567  -9.9301E-11   ... -2.9704E-08   2.1646E-09   5.7718E-11
```

To access the independent and dependent variables, you enter the following commands:

```
tablename.Independents
tablename.Dependents
```

The output looks like this:

```
>>> rdt.Independents
             Node

0               1
1               2
2               3
...
3708         3709
3709         3710
3710         3711
>>> rdt.Dependents
                X            Y            Z           XY           YZ           XZ
          (m)/(m)      (m)/(m)      (m)/(m)      (m)/(m)      (m)/(m)      (m)/(m)
0      7.9836E-14   4.5411E-14  -4.2656E-14  -6.7186E-13   8.8667E-13   3.1485E-13
1     -1.8053E-14   1.7968E-13  -9.1624E-14  -9.8358E-13   1.5946E-12   2.1604E-13
2     -2.4503E-13   1.5124E-13   1.0697E-13  -5.4891E-13   3.0055E-12  -1.7877E-13
...
3708  -3.8003E-10  -1.5106E-08   1.8513E-09  -2.2412E-08   1.2737E-08  -3.4123E-10
3709  -1.8019E-10  -1.0591E-08   1.2588E-09  -2.4495E-08   8.8607E-09   -2.409E-10
3710  -2.2277E-10  -7.2319E-09   8.5685E-10  -2.7292E-08   6.2061E-09  -2.2471E-10
```

To display a column's count and units for the dependants of the table, you enter this command:

```
tablename[columnName]
```

The output looks like this:

```
>>> rdt["XY"]
0               0
1               0
2               0
...
573        1.5708
574        1.5708
575        1.5708
Count: 576, Unit: "(rad)"
```

You can retrieve the unit and the count of the column values using this snippet:

```
value = tablename[columnName]
value.Unit
value.Count
```

The output looks like this:

```
    res = result1.PlotData
>>> value=res["X"]
>>> value.Unit
'(Pa)'
>>> value.Count
2368
```

Similarly, to display a specific contour result, you enter this command

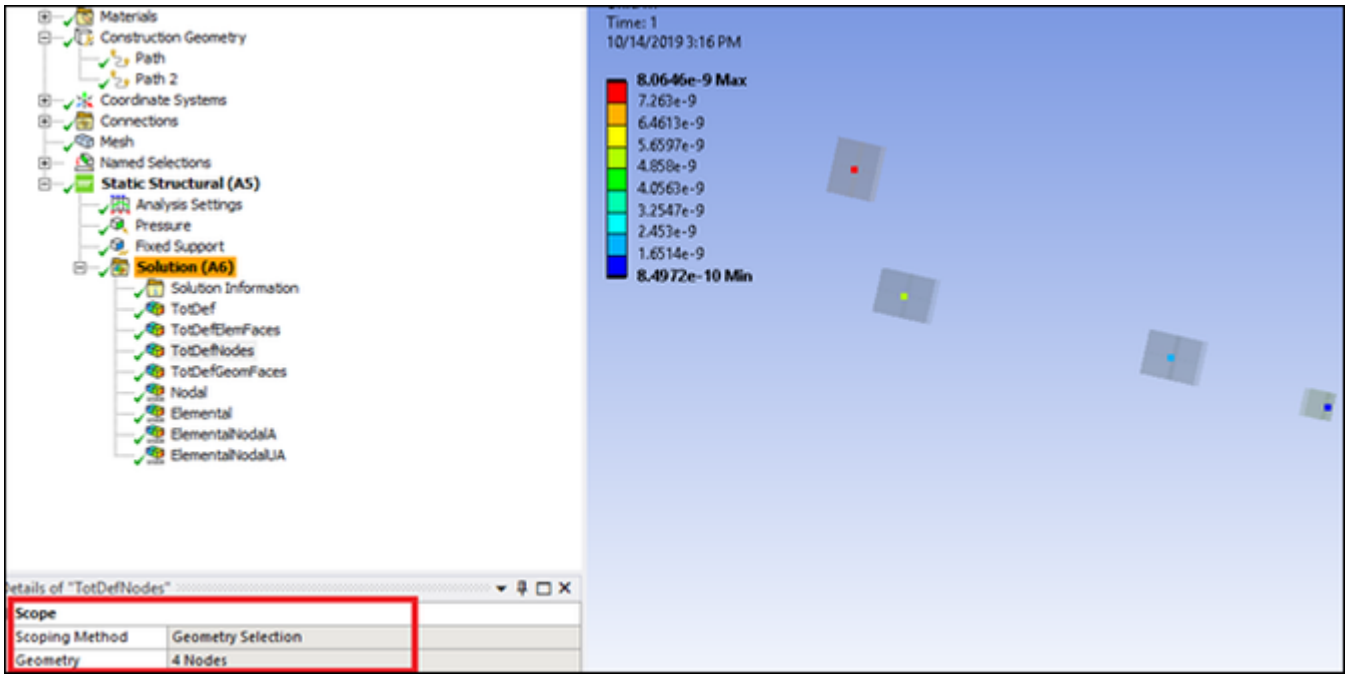
```
tablename[columnName][index]
```

The output looks like this:

```
>>> rdt["VOLUME"][5]
4.6875002226443E-06
```

Accessing Contour Results Scoped to Faces, Elements, or Nodes

You can scope contour results to an element face, a geometry face, or elements or nodes:



After the contour results are evaluated, the following code accesses them in the tabular data interface:

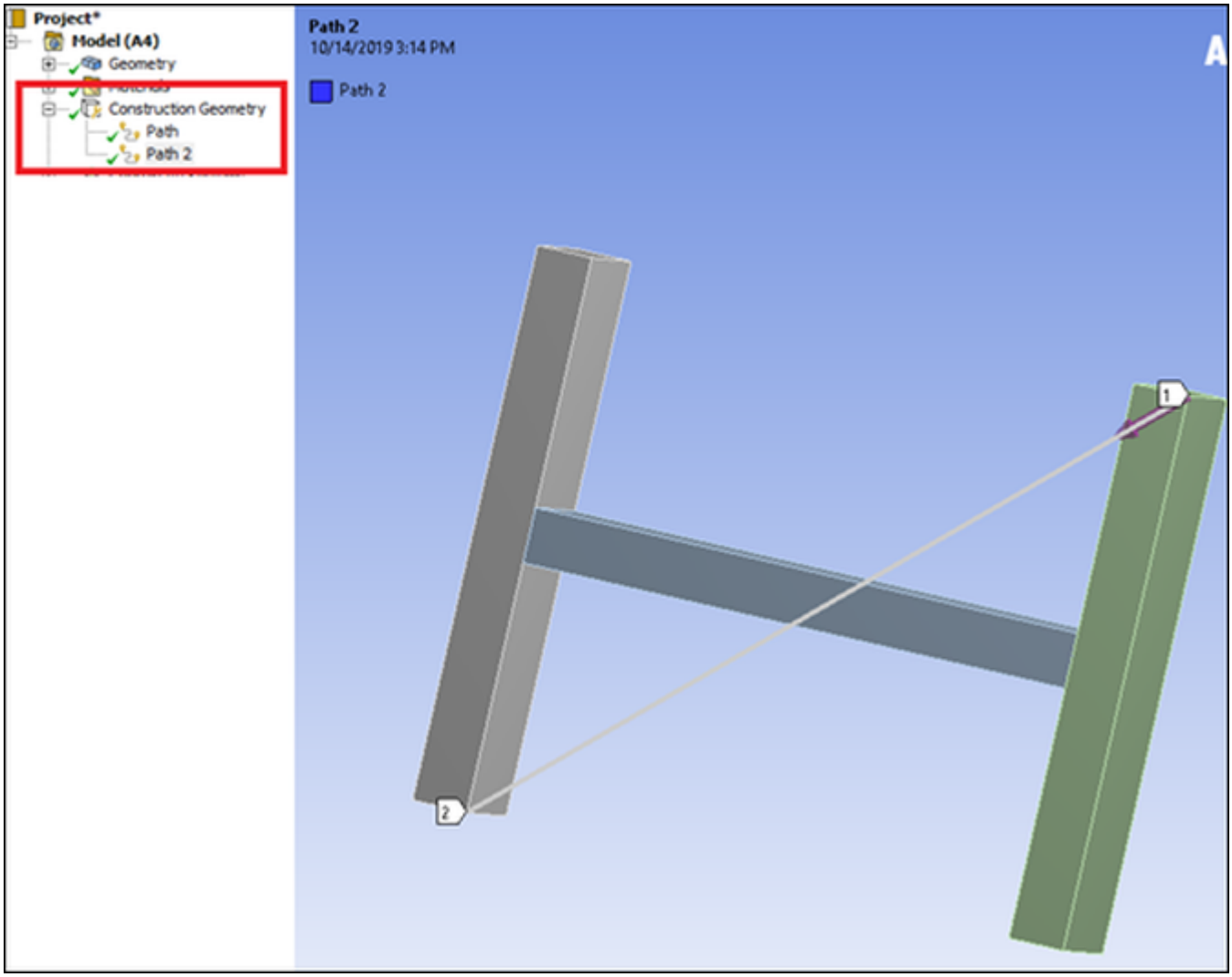
```
# Total Deformation Scoped to Nodes
print(" Result 3 : Total Deformation Nodes")
result3 = Model.Analyses[0].Solution.Children[3]
result3.PlotData
```

The following format applies for results scoped to a geometry face, element face, or a vertex:

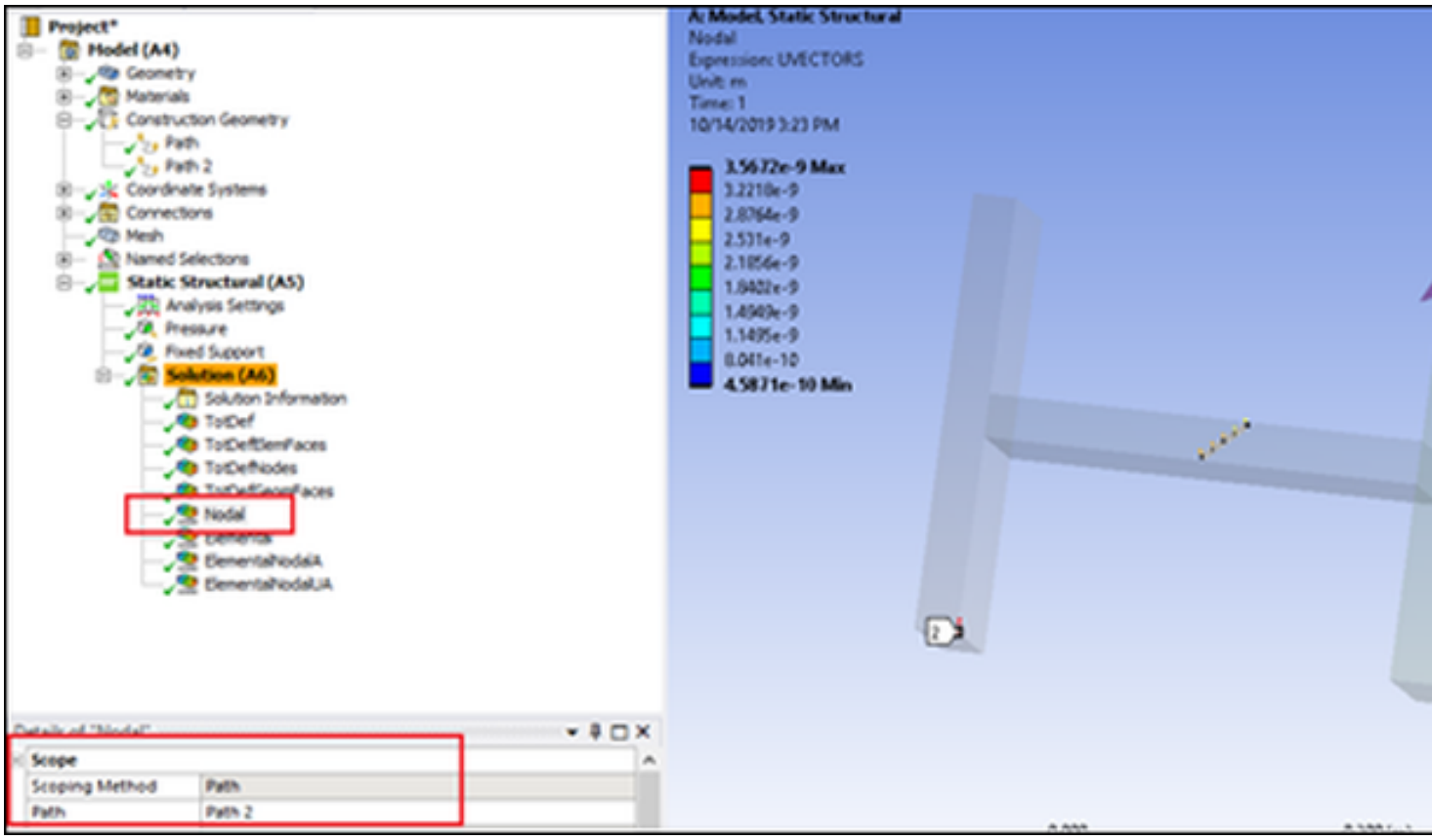
```
Result 3 : Total Deformation Nodes
          Node  TotDefNodes
                          m
0          327   8.0646E-09
1         1554   5.2589E-09
2         1564   1.9179E-09
3         3445   8.4972E-10
```

Accessing Contour Results Scoped to Paths

You can scope contour results to a path. Assume that the following path has been created:



You can scope the results to this path:



After the contour results are evaluated, the following code accesses them in the tabular data interface:

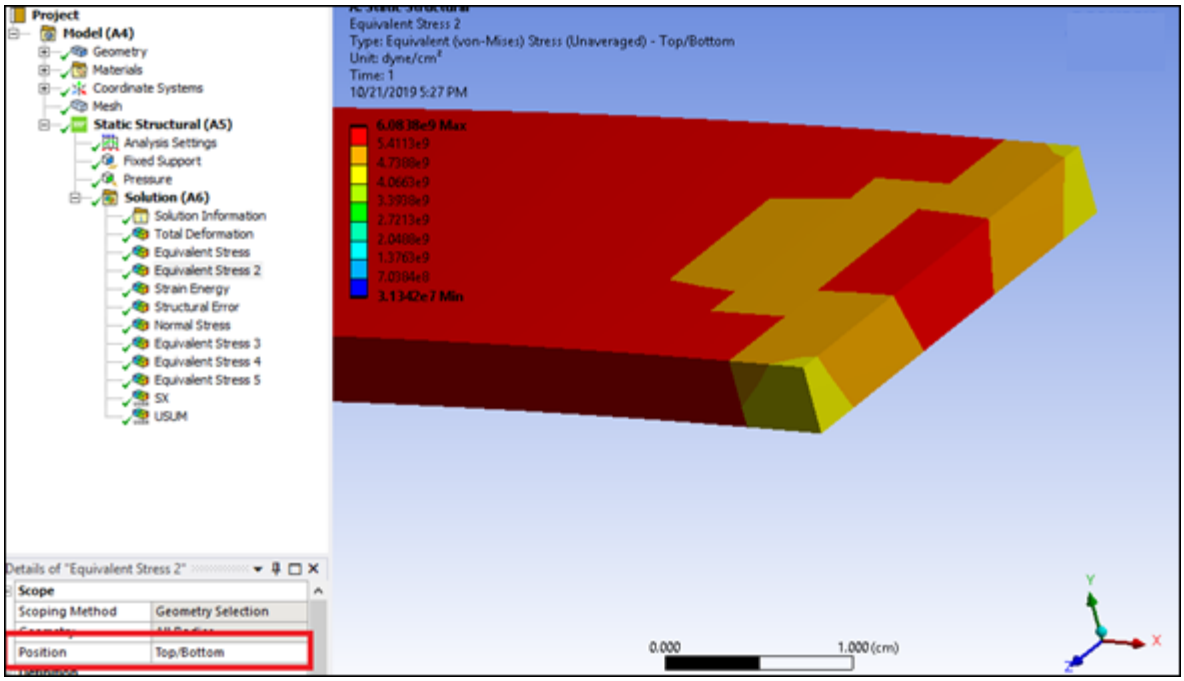
```
print(" Result 5 : Nodal")
result5 = Model.Analyses[0].Solution.Children[5]
result5.PlotData
```

The result is scoped to the X, Y, and Z coordinates of the path sampling points:

```
  Result 5 : Nodal
     X Coordi...  Y Coordi...  Z Coordi...           X           Y           Z
                                                      m           m           m
0            0.1           0         0.36  -9.2086E-11  -3.9368E-10  -2.1669E-10
1            0.1     0.00625     0.353125   -1.009E-10  -4.1399E-10  -2.3538E-10
2            0.1      0.0125      0.34625  -1.1019E-10  -4.3383E-10  -2.5442E-10
...
46           0.1      0.2875      0.04375  1.7977E+308  1.7977E+308  1.7977E+308
47           0.1     0.29375     0.036875  1.7977E+308  1.7977E+308  1.7977E+308
48           0.1         0.3         0.03   5.1201E-26  -3.5672E-09   1.1489E-23
```

Accessing Contour Results for Shells

For shell results, you can extract contour result values based on the shell side or position:



The following code accesses the evaluated contour results in the tabular data interface:

```
result = Model.Analyses[0].Solution.Children[3]
result.PlotData
```

The output looks like this:

	Node	Element	Shell Side	Equivale... (Pa)
0	1	113	Bottom	463358112
1	1	113	Top	447422272
2	2	25	Bottom	479172352
...				
1517	234	80	Top	528303520
1518	234	85	Bottom	514760064
1519	234	86	Top	517266304

When **`Position`** is set to **`onlyTop`**, **`onlyBottom`**, or **`onlyMiddle`**, the output has two values per node:

	Node	Shell Side	Equivale... (Pa)
0	1	Membrane	1.5306E+08
1	1	Membrane	1.5306E+08
2	2	Membrane	5.8305E+07
...			
465	233	Membrane	1.7673E+07
466	234	Membrane	9.853E+06
467	234	Membrane	9.853E+06

Limitations of Tabular Data Interface

With the tabular data interface, there are a few limitations:

- Surface results and beam results are not supported.
- Results reflect how the application stores the result values, which means that they are not converted to the user-specified unit system.
- Path results do not display the length across the path of the different points of the path (S parameter). However, you can retrieve this information by exporting the results to a text file.
- If the path does not have results in all of the nodes (such as through a hole), the empty spaces currently display huge values (DBL_MAX).
- List slicing is not supported on column results.
- When the result is scoped to more than one body, nodes shared by bodies might appear as multiple entries in the table, with the same or different result values. It may not be possible to determine the body for a given entry because no such information is presented in the table.

Other APIs

The following topics describe other Mechanical APIs of particular interest:

Mechanical Interface and Ribbon Tab Manipulation

UserInterface provides some control of the Mechanical user interface. This API is available using the following entry point:

```
ExtAPI.UserInterface
```

It allows you to control the ACT development, Button Editor, and ACT extension defined ribbon tabs. It cannot be used to control other toolbars or to create new buttons, ribbon tabs, or toolbars.

`ExtAPI.UserInterface.Toolbars` is a collection of toolbar objects.

- Each object has fields such as **`Name`**, **`Caption`**, **`Visibility`**, and **`child`** to access entries.
- Each child has the following properties: **`Caption`**, **`Enabled`**, **`Entries`**, **`EntryType`**, **`Name`**, and **`Visible`**.

The Boolean fields **`Visible`** and **`Enabled`** can be set to **`show`** or **`hide`** so that you can control the availability of the buttons depending on the current context.

Command Snippets

CommandSnippet provides control of the Commands object. You use the method **`AddCommandSnippet()`** to insert a new child command snippet in the project tree:

```
sol = Model.Analyses[0].Solution
cs = sol.AddCommandSnippet()
cs.Input = "/COM, New input"
cs.AppendText("\n/POST1")
```

You can also use **`ImportTextFile(string`** to import content from a text file or use **`ExportTextFile(string)`** to export a command snippet to a text file.

Object Tags

You can use APIs to access object tags and to add or remove them:

To access object tags:

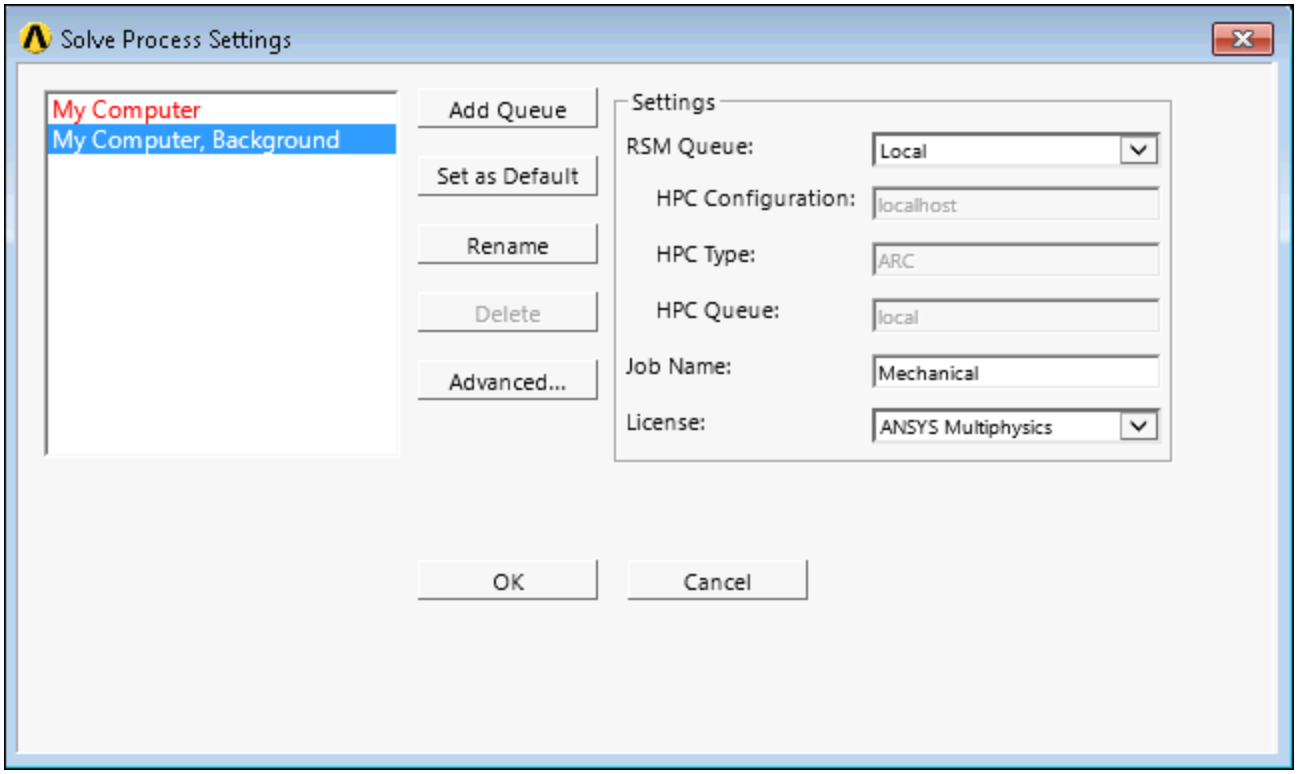
```
tag2 = DataModel.ObjectTags[2]
for tag in DataModel.ObjectTags:
name = tag.Nam
objects = tag.Objects
```

To add or remove object tags:

```
tag = Ansys.Mechanical.Application.ObjectTag("supports")
tag.Objects = [obj1, obj2, obj3, ...]
DataModel.ObjectTags.Add(tag)
DataModel.ObjectTags.Remove( Ansys.Mechanical.Application.ObjectTag("loads")
```

Solve Process Settings

The Solve Process Settings API is defined using two collections, a read-only collection of RSMQueue and a mutable collection of SolveConfiguration. The **RSMQueue** collection lets you inspect the information on the right side of the **Solve Process Settings** dialog box for each possible selection in the **RSM Queue** property. Each item in the field on the left side is a **SolveConfiguration**. Using the collection API, you can edit the collection or individual entities in the collection.



This script accesses and changes some details from the second solve configuration:

```
config2 = ExtAPI.Application.SolveConfigurations["My Computer, Background"]
x = config2.Default
y = config2.Settings.License
```

```
z = config2.SolveProcessSettings.ManualSolverMemorySettings.Workspace
config2.SolveProcessSettings.ManualLinuxSettings.UserName = "jane.doe"
config2.SolveProcessSettings.MaxNumberOfCores = 12
```

This script accesses some details from the first RSM queue:

```
queue1 = ExtAPI.Application.RSMQueues[0]
x = queue1.Name
y = queue1.HPCConfiguration
```

This script modifies the collection of solve configurations and solve:

```
collection = ExtAPI.Application.SolveConfigurations
new_config = Ansys.ACT.Mechanical.Application.SolveProcessSettings.SolveConfiguration()
new_config.Name = "My cluster"
collection.Add(new_config)
new_config.SetAsDefault()
new_config.Settings.License = "Mechanical Enterprise"
analysis.Solve()
```

Message Window

The Message window API uses a collection called Messages. You can use this API to access individual messages and their data and operate on the collection to add and remove messages:

- Access a messages and its data:

```
thirdMsg = ExtAPI.Application.Messages[2]
for msg in ExtAPI.Application.Messages:
    obj = msg.Source
    stringId = msg.StringID
    caption = msg.DisplayString
    scoping = msg.Location
    time = msg.TimeStamp
    objs = msg.RelatedObjects
    severity = msg.Severity
```

- Add a message:

```
msg = Ansys.Mechanical.Application.Message("Problem!", MessageSeverityType.Error)
ExtAPI.Application.Messages.Add(msg)
```

Interacting with Legacy JScript

When used from JScript that is executed from the Mechanical APIs, the JScript function **`returnFromScript()`** allows you return values from JScript back to Python or C#

In the following code example, nothing is returned because **`returnFromScript()`** is not used:

```
x = ExtAPI.Application.ScriptByName("jscript").ExecuteCommand("""var x=1""")
```

However, if you use the function **`returnFromScript()`**, you can return the value of x from the JScript back to the Python code:

```
x = ExtAPI.Application.ScriptByName("jscript").ExecuteCommand("""var     x=1
returnFromScript(x)""")
```

In this case, x now holds the value 1 because this value was passed into the Python code from the **`returnFromScript()`** function.

Supported Return Types

The function **`returnFromScript()`** can return values from the following data types:

- Int
- Double
- Boolean
- String
- Lists (Windows only)
- Dictionaries (Windows only) if created as Automation objects:

```
var dict = new ActiveXObject("Scripting.Dictionary");
```

Known Limitations

The following limitations exist:

- A list can hold a maximum of 65,535 items.
- Lists and dictionaries do not work on Linux.
- In the JScript that is to be executed using **`ExecuteCommand()`**, **`ret`** is a reserved keyword.

Example

The JScript that follows includes many functions for returning different types of variables:

```
script = """
     function returnInt(){
         return 10;
     }
 Function returnDouble(){
       return 3.14;
      }
     function returnString(){
         return "Testing J Script";
     }
     function returnBool(){
         return true;
     }
     var innerDict = new ActiveXObject("Scripting.Dictionary");

     function returnList(){
         var x = 10;
         var y = 20;
         var str = "thirty"
         var nullVal = null;
         var boolVal = true;
         var innerList1 = [1 ,2, 3, "str", [1, "str", 2]];
         var innerList2 = [[1], [1,2,3,[1,5,6]]];
  innerDict.Add("Testing", 1);
```

```
        var list = [x,y, str, nullVal, boolVal, innerList1,
 innerList2, innerDict]
        return list;
    }

 function returnLongList(){
        var longList = [];
        for(var i= 0; i<65535; i++){
            longList[i] = i;
        }

        return longList;
    }

    function returnDict(){
        var dict = new ActiveXObject("Scripting.Dictionary");
        var listTry = [1,2,3]
        dict.Add("string", "strTest");
        dict.Add("int", 1);
        dict.Add("bool", true);
        dict.Add("null", null);
        dict.Add("list", [1,2,3])
        dict.Add(1, "int");
        dict.Add(true, "bool");
        dict.Add([1,2,3], "list");

        innerDict.Add("Testing", 1);
        dict.Add("dict", innerDict);

        return dict;
    }
    var retVal = returnInt();
    returnFromScript(retVal );

""")
```

As demonstrated, you can set the variable **`retVal`** to whatever is returned from some function. You can then pass the variable **`retVal`** into **`returnFromScript()`** to return it to the Python code.

Return a list

The function **`returnList()`** can be used as a reference when returning lists from the JScript code.

Return a dictionary

The function **`returnDict()`** can be used as a reference when returning dictionaries from the JScript code.

Passing in a value to the JScript code from Python code

To pass in a value to the JScript code from Python code, you embed string values of Python code in the script.

The goal of this first example is to pass an integer value (5) to the JScript code from the Python code and then increment it by 1 in the JScript code and return the new value (6) to the Python code.

```
x = 5
script = """
var x = """ + str(x) + """;
x++;
returnFromScript(x);
"""
x = ExtAPI.Application.ScriptByName("jscript").ExecuteCommand(script)
```

The goal of this second example is to pass a list (1, 2, 3, 5) to the JScript code from the Python code and then update the fourth element and return the updated list (1, 2, 3, 10) to the Python code.

```
x = [1, 2, 3, 5]
script = """
var x = """ + str(x) + """;
x[3] = 10;
returnFromScript(x);
"""
x = ExtAPI.Application.ScriptByName("jscript").ExecuteCommand(script)
```

Note:

To pass in a Boolean value from Python to JScript, you must first convert it to a string and then make the string all lowercase because Booleans in Python start with an uppercase character where in JScript they start with a lowercase character.

Scripting Examples

This section provides many examples of scripts that you can use to easily complete both common and novel tasks in Mechanical. Scripts are organized by three main categories, though some scripts overlap categories.

When logged into your ANSYS customer account, you can search solutions for additional script examples using the following filter selections:

- For **Product**, select **ACT Customization Suite**.
- For **Product Family**, select **Scripting**.

To further limit the solutions shown, you can use the search box at the top of the page.

Script Examples for Selection

The following scripts are for making selections in Mechanical:

Select Geometry or Mesh in the Graphics Window

Goal: Select a geometry or mesh in the graphics window.

Code for Geometry selection:

```
# Clear the current selection and select some previously determined Geo IDs
ExtAPI.SelectionManager.ClearSelection()
mySel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
mySel.Ids = [17,19,22]   #These are the IDs of any geometric entities
ExtAPI.SelectionManager.NewSelection(mySel)
```

Code for Mesh selection:

You might need to be in the wireframe mode to see the selected nodes.

```
# Clear the current selection and select some mesh nodes
ExtAPI.SelectionManager.ClearSelection()
mySel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.MeshNodes)
mySel.Ids = [1,2,3,4]  # These are the IDs of any node entities
ExtAPI.SelectionManager.NewSelection(mySel)
```

Get Tree Object of a Body Corresponding to a Selected Body

Goal: Get the tree object corresponding to a selected body in the graphics window.

Code:

```
myIds = ExtAPI.SelectionManager.CurrentSelection.Ids
geoBody = DataModel.GeoData.GeoEntityById(myIds[0])
treeBody = Model.Geometry.GetBody(geoBody)
```

Get GeoData Body Corresponding to a Tree Object of a Body

Goal: Get the GeoData (graphics window) body corresponding to a tree object of a body.

Code:

```
treeBody = Model.Geometry.Children[0].Children[0]
geoBody = treeBody.GetGeoBody()
```

Query Mesh Information for Active Selection

Goal: Query the mesh information for the active selection.

Code:

```
# Macro to demonstrate how to query mesh information for the active selection

meshData = DataModel.MeshDataByName("Global")
sel=ExtAPI.SelectionManager.CurrentSelection
refIds = sel.Ids

# check that current selection is nodes
if sel.SelectionType == SelectionTypeEnum.MeshNodes:
    print "%s Nodes Selected" % (refIds.Count)
    for meshId in refIds:
        node = meshData.NodeById(meshId)
        print "Node %s, X = %s, Y = %s, Z = %s" % (meshId, node.X, node.Y, node.Z)
else:
    print "Selection is not made of nodes"
```

Use an Existing Graphics Selection on a Result Object

Goal: Take an existing graphics selection and use it on a result object.

This example works best when selecting nodes and elements while already viewing a result.

Code:

```
mysel = ExtAPI.SelectionManager.CurrentSelection
myres = Tree.FirstActiveObject
myres.ClearGeneratedData()
myres.Location = mysel
myres.EvaluateAllResults()
```

Calculate Sum of Volume, Area, and Length of Scoped Entities

Goal: Calculate the sum of the volume, area, and length of scoped entities.

Code:

```
sum = 0

for geoid in  ExtAPI.SelectionManager.CurrentSelection.Ids  :
    geoEntity = DataModel.GeoData.GeoEntityById(geoid)
    if geoEntity.Type == GeoCellTypeEnum.GeoBody:
        sum += geoEntity.Volume
        type = "volume"
    if geoEntity.Type == GeoCellTypeEnum.GeoFace:
        sum += geoEntity.Area
        type = "area"
    if geoEntity.Type == GeoCellTypeEnum.GeoEdge:
        sum += geoEntity.Length
        type = "length"

# values are reported in the CAD unit system so get that
unit = Model.Geometry.LengthUnit
print("Total selected "+ type + " is: " + str(sum) + " "+ str(unit))
```

Create a Named Selection from the Scoping of a Group of Objects

Goal: Loop over all contacts in the tree and create a single named selection scoped to all of the faces.

Code:

```
selmgr=ExtAPI.SelectionManager
selmgr.ClearSelection()
contacts = DataModel.GetObjectsByType(DataModelObjectCategory.ContactRegion)
for contact in contacts:
    selmgr.AddSelection(contact.SourceLocation)
    selmgr.AddSelection(contact.TargetLocation)

total=selmgr.CurrentSelection.Ids.Count

Model.AddNamedSelection()

print "Done with macro, Create a NS with %s selections" % (total)
```

Create a Named Selection that Selects All Faces at a Specified Location

Goal: Create a named selection that selects all faces at the zero location of the XY and XZ planes.

Code:

```
NS1 = DataModel.Project.Model.AddNamedSelection()
NS1.ScopingMethod = GeometryDefineByType.Worksheet
GenerationCriteria = NS1.GenerationCriteria
Criterion1 = Ansys.ACT.Automation.Mechanical.NamedSelectionCriterion()
Criterion1.Action = SelectionActionType.Add
Criterion1.EntityType = SelectionType.GeoFace
Criterion1.Criterion = SelectionCriterionType.LocationY
Criterion1.Operator = SelectionOperatorType.Equal
Criterion1.Value = Quantity("0 [m]")
GenerationCriteria.Add(Criterion1)
Criterion2 = Ansys.ACT.Automation.Mechanical.NamedSelectionCriterion()
Criterion2.Action = SelectionActionType.Add
Criterion2.EntityType = SelectionType.GeoFace
```

```
Criterion2.Criterion = SelectionCriterionType.LocationZ
Criterion2.Operator = SelectionOperatorType.Equal
Criterion2.Value = Quantity("0 [m]")
GenerationCriteria.Add(Criterion2)
NS1.Generate()
```

Rescope a Solved Result Based on the Active Node or Element Selection

Goal: For a solved result, take the current selection in the graphics window (of either nodes or elements) and create a copy of the result scoped to this selection.

Code:

```
selmgr = ExtAPI.SelectionManager
loc = selmgr.CurrentSelection

res = Tree.FirstActiveObject

# verify object is a result or a custom result
isRegularResult = isinstance(res, Ansys.ACT.Automation.Mechanical.Results.Result)
isCustomResult = res.DataModelObjectCategory == DataModelObjectCategory.UserDefinedResult

if (isRegularResult or isCustomResult):

    newRes = res.Duplicate()
    newRes.ClearGeneratedData()
    newRes.Location=loc
    newRes.EvaluateAllResults()

else:
    print "Selected Object is not a Result!"
```

Scope a Boundary Condition to a Named Selection

Goal: Use a macro to create a fixed support and then scope it to a named selection.

Code:

```
# Macro to demonstrate the ability to create a fixed support and then scope it to a Named Selection
#
ns = DataModel.GetObjectsByName("myFaces")
# make sure only 1 named selection was found
if (ns.Count != 1):
    print("A single Named Selection was not Found!")
else:

    # get the first analsysis and make sure it is structural
    analysis = DataModel.AnalysisList[0]

    if (analysis.PhysicsType != PhysicsType.Mechanical):
        print("The first analysis isn't structural")
    else:
        # Finally create a fixed support and scope it to the Named Selection
        mySupport = analysis.AddFixedSupport()
        mySupport.Location=ns[0]
        print("Done creating a Fixed Support and Scoping to a Named Selection")
```

Add a Joint Based on Proximity of Two Named Selections

Goal: Given a named selection of *N* faces and another of *N*2* faces, add a joint between each face in the first named selection and the nearest two faces in the second named selection.

Code:

```
import math

#distance formula where c1 and c2 are each arrays of 3 real numbers
def dist(c1, c2):
    x = c1[0]-c2[0]
    x = x*x
    y = c1[1]-c2[1]
    y = y*y
    z = c1[2]-c2[2]
    z = z*z
    return math.sqrt(x+y+z)

jh1 = DataModel.GetObjectsByName("jh1")[0]       #named selection object of faces to become the single joint refer
jh2 = DataModel.GetObjectsByName("jh2")[0]       #named selection object of faces from which the two closest will
group = DataModel.GetObjectsByName("Connection Group")[0]       #connection group to hold all of the joints that a

geo = DataModel.GeoData
face_ids1 = jh1.Location.Ids       #face ids in jh1
face_ids2 = jh2.Location.Ids       #face ids in jh2

#get the centroids of all faces in face_ids2
face_centroids2 = [geo.GeoEntityById(face_id2).Centroid for face_id2 in face_ids2]

def get_min_indeces(face_centroid1):
    #array to hold the index and distance to face1_center for two closest faces in face_centroids2
    min_indeces = [None, None]

    for index in range(len(face_centroids2)):
        face_centroid2 = face_centroids2[index]
        distance = dist(face_centroid1, face_centroid2)

        #initilize with the first two iterations of this loop
        if index == 0:
            min_indeces[0] = (index, distance)
            continue
        if index == 1:
            min_indeces[1] = (index, distance)
            continue

        #get the index of the item with the larger distance
        if min_indeces[0][1] < min_indeces[1][1]:
            larger_dist_index = 1
        else:
            larger_dist_index = 0

        #replace that item with the current face if the distance is smaller
        if distance < min_indeces[larger_dist_index][1]:
            min_indeces[larger_dist_index] = (index, distance)
    return min_indeces

#use a transaction to speed up adding joints to the tree in a loop
with Transaction():
    for face_id1 in face_ids1:
        face1 = geo.GeoEntityById(face_id1)
        face_centroid1 = face1.Centroid       #get the centroid of each face in face_ids1
        min_indeces = get_min_indeces(face_centroid1)

        #get the face ids using the indices computed above
        face_ids = [face_ids2[min_indeces[0][0]], face_ids2[min_indeces[1][0]]]

        #add a joint and assign its geometry selection
        joint = group.AddJoint()
```

```
        reference_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
        reference_selection.Ids = [face_id1]

        mobile_selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
        mobile_selection.Ids = face_ids

        joint.ReferenceLocation = reference_selection
        joint.MobileLocation = mobile_selection
```

Print Selected Element Faces

Goal: Print the element id and face index of all selected element faces.

Code:

```
# Purpose of the script: prints selected element faces area

g_elementTypeToElemFaceNodeIndices = {
    ElementTypeEnum.kTri3 : [ [ 0, 1, 2 ] ],
    ElementTypeEnum.kTri6 : [ [ 0, 1, 2, 4, 5, 6 ] ],

    ElementTypeEnum.kQuad4 : [ [ 0, 1, 2, 4 ] ],
    ElementTypeEnum.kQuad8 : [ [ 0, 1, 2, 4, 5, 6, 7, 8 ] ],

    ElementTypeEnum.kTet4 : [ [ 0, 1, 3 ], [ 1, 2, 3 ], [ 2, 0, 3 ], [ 0, 2, 1 ] ],
    ElementTypeEnum.kTet10 : [ [ 0, 1, 3, 4, 8, 7 ], [ 1, 2, 3, 5, 9, 8 ], [ 2, 0, 3, 6, 7, 9 ], [ 0, 2, 1, 6, 5,

    ElementTypeEnum.kPyramid5 : [ [ 0, 3, 2, 1 ], [ 0, 1, 4 ], [ 1, 2, 4 ], [ 3, 4, 2 ], [ 0, 4, 3 ] ],
    ElementTypeEnum.kPyramid13 : [ [ 0, 3, 2, 1, 8, 7, 6, 5 ], [ 0, 1, 4, 5, 10, 9 ], [ 1, 2, 4, 6, 11, 10 ], [ 3,

    ElementTypeEnum.kWedge6 : [ [ 0, 2, 1 ], [ 3, 4, 5 ], [ 0, 1, 4, 3 ], [ 1, 2, 5, 4 ], [ 0, 3, 5, 2 ] ],
    ElementTypeEnum.kWedge15 : [ [ 0, 2, 1, 8, 7, 6 ], [ 3, 4, 5, 9, 10, 11 ], [ 0, 1, 4, 3, 6, 13, 9, 12 ], [ 1,

    ElementTypeEnum.kHex8 : [ [ 0, 1, 5, 4 ], [ 1, 2, 6, 5 ], [ 2, 3, 7, 6 ], [ 3, 0, 4, 7 ], [ 0, 3, 2, 1 ], [ 4,
    ElementTypeEnum.kHex20 : [ [ 0, 1, 5, 4, 8, 17, 12, 16 ], [ 1, 2, 6, 5, 9, 18, 13, 17 ], [ 2, 3, 7, 6, 10, 19,
}

def Norm(vec):
    return sqrt(vec[0]*vec[0] + vec[1]*vec[1] + vec[2]*vec[2])

def CrossProduct(a, b):
    return [ a[1]*b[2] - a[2]*b[1], a[2]*b[0] - a[0]*b[2], a[0]*b[1] - a[1]*b[0] ]

def TriangleArea(a, b, c):
    ab = [b[0]-a[0], b[1]-a[1], b[2]-a[2]]
    ac = [c[0]-a[0], c[1]-a[1], c[2]-a[2]]
    n = CrossProduct(ab, ac)
    area = 0.5 * Norm(n)
    #print("a={0}, b={1}, c={2}, ab={3}, ac={4}, n={5}, area={6}".format(a, b, c, ab, ac, n, area))
    return area

def GetElementFaceNodes(element_id, element_face_index):
    mesh = ExtAPI.DataModel.MeshDataByName("Global")
    element = mesh.ElementById(element_id)
    element_node_indices = g_elementTypeToElemFaceNodeIndices[element.Type][element_face_index]
    return [element.Nodes[element_node_index] for element_node_index in element_node_indices]

def GetElementFaceArea(element_id, element_face_index):
    element_face_nodes = GetElementFaceNodes(element_id, element_face_index)
    if len(element_face_nodes) == 3 or len(element_face_nodes) == 6:
        # it's a triangle
        return \
        TriangleArea( \
            [element_face_nodes[0].X, element_face_nodes[0].Y, element_face_nodes[0].Z], \
            [element_face_nodes[1].X, element_face_nodes[1].Y, element_face_nodes[1].Z], \
            [element_face_nodes[2].X, element_face_nodes[2].Y, element_face_nodes[2].Z])
    else:
        # it's a quadrangle (made of 2 triangles)
```

```
        return \
        TriangleArea( \
            [element_face_nodes[0].X, element_face_nodes[0].Y, element_face_nodes[0].Z], \
            [element_face_nodes[1].X, element_face_nodes[1].Y, element_face_nodes[1].Z], \
            [element_face_nodes[2].X, element_face_nodes[2].Y, element_face_nodes[2].Z]) + \
        TriangleArea( \
            [element_face_nodes[0].X, element_face_nodes[0].Y, element_face_nodes[0].Z], \
            [element_face_nodes[2].X, element_face_nodes[2].Y, element_face_nodes[2].Z], \
            [element_face_nodes[3].X, element_face_nodes[3].Y, element_face_nodes[3].Z])

def GetCurrentSelectedElementFaces():
    current_selection = ExtAPI.SelectionManager.CurrentSelection
    if current_selection.SelectionType == SelectionTypeEnum.MeshElementFaces:
        element_ids = current_selection.Ids
        element_face_indices = current_selection.ElementFaceIndices
        return (element_ids, element_face_indices)
    elif current_selection.SelectionType == SelectionTypeEnum.MeshElements:
        element_ids = current_selection.Ids
        element_face_indices = [0 for i in element_ids]
        return (element_ids, element_face_indices)
    else:
        return ([], [])

def PrintCurrentSelectedElementFacesElementFacesArea():
    (element_ids, element_face_indices) = GetCurrentSelectedElementFaces()
    if len(element_ids) < 1:
        print('No element faces selected')
        return

    text = ''
    for i in range(len(element_ids)):
        area = GetElementFaceArea(element_ids[i], element_face_indices[i])
        text += 'element id={0}, faceIndex={1}, area={2}\n'.format(element_ids[i], element_face_indices[i], area)
    print(text)

PrintCurrentSelectedElementFacesElementFacesArea()
```

Get Normal of a Face

Goal: Print the normal of a given face at a given location in space.

Code:

```
face_id = 20 #Current selection in ExtAPI.SelectionManager.CurrentSelection
point = (.01,.015,0.) #point in xyz space in the CAD unit system

#get the face object
face = DataModel.GeoData.GeoEntityById(face_id)

#get the projected point on the geometry (a curvilinar abscissa for an edge, (u,v) for a face)
u,v = face.ParamAtPoint(point)

#print the normal at the point
print(face.NormalAtParam(param))
```

Create a Selection Based on the Location of Nodes in Y

Goal: Given a Y location and tolerance, create a selection of all nodes within that tolerance from their global Y coordinate location.

Code:

```
# Get access to mesh
mesh = DataModel.MeshDataByName(ExtAPI.DataModel.MeshDataNames[0])
```

```
selected_node_ids = set() # Empty set
for node in mesh.Nodes:
    tolerance = 0.0001 # 0.1 mm
    y_location = 0.0130
    if (node.Y >= y_location - tolerance) and (node.Y <= y_location + tolerance):
        selected_node_ids.add(node.Id)

# Create selection
selection_info = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.MeshNodes)
for selected_node_id in selected_node_ids:
    selection_info.Ids.Add(selected_node_id)

ExtAPI.SelectionManager.NewSelection(selection_info)
```

Create Aligned Coordinate Systems in a Motor

Goal: Given a motor model with a named selection of faces and two named selections of vertices, all of the same size, for each of the three lists, add a coordinate system at the center of the face with its +Y axis pointed out of the face. The -X direction will be from the vertex in the second list to the vertex in the third list.

Note:

The following script example is model-specific. Click here to download the archived ANSYS project.

Code:

```
# Get all faces and vertices of named selections
face_named_selection = Model.NamedSelections.Children[0] # First named selection, of faces
face_ids = face_named_selection.Ids
start_vertex_named_selection = Model.NamedSelections.Children[1] # Second named selection, of vertices
start_vertex_ids = start_vertex_named_selection.Ids
end_vertex_named_selection = Model.NamedSelections.Children[2] # Third named selection, of vertices
end_vertex_ids = end_vertex_named_selection.Ids

# Create axis systems
selection_info = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
with Transaction(): # Suppress tree update for performance
    for iFaceCount in range(0, face_ids.Length):
        coordinate_system = Model.CoordinateSystems.AddCoordinateSystem()
        selection_info.Ids.Clear()
        selection_info.Ids.Add(face_ids[iFaceCount]) # Create a temporary selection

        # Define origin in face center
        coordinate_system.OriginLocation = selection_info

        # Define primary axis (Y) normal into face
        coordinate_system.PrimaryAxis = CoordinateSystemAxisType.PositiveYAxis
        coordinate_system.PrimaryAxisDefineBy = CoordinateSystemAlignmentType.Associative # Geometry selection
        coordinate_system.PrimaryAxisLocation = selection_info

        # Define orientation around principle axis from start- to endvertex
        selection_info.Ids.Clear()
        selection_info.Ids.Add(end_vertex_ids[iFaceCount]) # Create a temporary selection
        selection_info.Ids.Add(start_vertex_ids[iFaceCount]) # Create a temporary selection
        coordinate_system.SecondaryAxis = CoordinateSystemAxisType.PositiveXAxis
        coordinate_system.SecondaryAxisDefineBy = CoordinateSystemAlignmentType.Associative # Geometry selection
        coordinate_system.SecondaryAxisLocation = selection_info
```

Script Examples for Interacting with Tree Objects

The following scripts are for interacting with tree objects in Mechanical:

Delete an Object

Goal: Delete a selected object from the tree.

Code:

```
ObjToDelete.Delete()
```

Refresh the Tree

Goal: Refresh the Mechanical tree. This is needed for certain operations that visually appear only after an update of the tree.

Code:

```
Tree.Refresh()
```

Get All Visible Properties for a Tree Object

Goal: Get all visible properties for a native tree object and then print their captions and string values.

Code:

```
force = Model.Analyses[0].AddForce()
for forceProperty in force.VisibleProperties:
    print(forceProperty.Caption + " | " + forceProperty.StringValue)
```

Parametrize a Property for a Tree Object

Goal: Parametrize a **Details View** property for a native tree object and then print out the Workbench ID.

Code:

```
hydrostaticPressure = Model.Analyses[0].AddHydrostaticPressure()
fluidDensity = hydrostaticPressure.CreateParameter("FluidDensity")
fluidDensity.ID
```

Count the Number of Contacts

Goal: Count the number of contacts in the model.

Code:

```
contacts = DataModel.GetObjectsByType(DataModelObjectCategory.ContactRegion)
numContacts = contacts.Count
print("There are %s contact regions" % (numContacts) )
```

Verify Contact Size

Goal: For a face-to-face contact, ensure that all "contact" sides are smaller than their "target" sides.

Code:

```
geom=DataModel.GeoData

contacts = DataModel.GetObjectsByType(DataModelObjectCategory.ContactRegion)

with Transaction():
    for cont in contacts:
    source_area = 0
```

```
    sourcenum = cont.SourceLocation.Ids.Count
    for x in range(0,sourcenum):
        myface = geom.GeoEntityById(cont.SourceLocation.Ids[x])
        source_area = source_area + myface.Area

    target_area = 0
    targetnum = cont.TargetLocation.Ids.Count
    for x in range(0,targetnum):
        myface = geom.GeoEntityById(cont.TargetLocation.Ids[x])
        target_area = target_area + myface.Area

    if (target_area < source_area):
        print "Flipping Source/Target For Contact Region = %s" % (cont.Name)
        cont.FlipContactTarget()

print "Done with Macro"
```

Set Pinball to 5mm for all Frictionless Contacts

Goal: Change all frictionless contacts to have a pinball of 5mm.

Code:

```
with Transaction():
    contacts = DataModel.GetObjectsByType(DataModelObjectCategory.ContactRegion)
    changeCount = 0
    for cont in contacts:
        if (cont.ContactType == ContactType.Frictionless) :
            cont.PinballRegion = ContactPinballType.Radius
            cont.PinballRadius = Quantity("5[mm]")
            changeCount = changeCount + 1
print "Done with macro, changed %s contact regions" % (changeCount)
```

Use a Named Selection as Scoping of a Load or Support

Goal: Use a named selection as scoping of a load or support object.

Code:

```
load = Model.Analyses[0].AddPressure
ns = Model.NamedSelections.Children[0]
load.Location = ns
```

Suppress Bodies Contained in a Given Named Selection

Goal: Retrieve and suppress the bodies contained in a named selection.

Code:

```
ns = Model.NamedSelections.Children[0] # selected a named selection containing bodies
bodyIds = ns.Location.Ids
with Transaction():
    for bodyId in bodyIds:
        geoBody = DataModel.GeoData.GeoEntityById(bodyId)
        body = Model.Geometry.GetBody(geoBody)
        body.Suppressed = True
print "Done with Macro"
```

Modify the Scoping on a Group of Objects

Goal: Loop over the selected tree objects and remove the second item to which it is scoped. This example fixes a set of pretension bolt loads that were scoped to two faces of a split cylinder and needed to be scoped to only one face.

Code:

```
with Transaction():
    for obj in Tree.ActiveObjects:
        loc = obj.Location
        loc.Ids.RemoveAt(1)
        obj.Location = loc
```

Change Tabular Data Values of a Standard Load or Support

Goal: Change the tabular data values of a standard load or support.

Code:

```
pressureLoad = Model.Analyses[0].AddPressure()
pressureLoad.Magnitude.Output.DiscreteValues = [Quantity('0 [MPa]'),Quantity('10 [MPa]')]
```

Duplicate an Harmonic Result Object

Goal: Given a selected harmonic result, duplicate it and sweep over the phase.

Code:

```
# nDiv is the number of results you want to create on a 360 basis,
# so setting to 30 will create a result every 12 degrees(360/30)
nDiv = 30
angleInc = 360/nDiv

BaseResult = Tree.FirstActiveObject

for n in range(0,nDiv+1):
    angle = Quantity(str(n*angleInc) + ' [degree]')
    newResult = BaseResult.Duplicate()
    newResult.SweepingPhase = angle
    newResult.Name = 'Sweep At ' + str(n*angleInc)
```

Retrieve Object Details Using SolverData APIs

Goal: Retrieve global and object-specific solver data from a solved analysis using **`SolverData`** APIs.

Code:

```
#Solver data in a solved analysis
solution = Model.Analyses[0].Solution
solver_data = solution.SolverData
solver_data.MaxElementId
solver_data.MaxNodeId
solver_data.MaxElementType
```

```
#Body data in a solved analysis
geometry = Model.Geometry
base = [i for i in geometry.GetChildren[Ansys.ACT.Automation.Mechanical.Body](True) if i.Name == 'Base'][0]
body_data = solver_data.GetObjectData(base)
body_data.ElementTypeIds
body_data.MaterialIDs
body_data.RealConstantId
```

```
#Spring data in a solved analysis
connection_group = Model.Connections
spring = [i for i in connection_group.GetChildren[Ansys.ACT.Automation.Mechanical.Connections.Spring](True) if i.N
spring_data = solver_data.GetObjectData(spring)
spring_data.RealConstantId
spring_data.ElementId
```

```
#Force data in a solved analysis
static_structural = Model.Analyses[0]
force = static_structural.Children[4]
load_data = solver_data.GetObjectData(force)
load_data.SurfaceEffectElementTypeId
```

Evaluate Spring Reaction Forces

Goal: Evaluate spring reaction forces using the **`SolverData`** API and the result reader.

Code:

```
# Get access to solver data
analysis = Model.Analyses[0]
solver_data = analysis.Solution.SolverData

# Get access to result reader
with analysis.GetResultsData() as reader:
    spring_results = reader.GetResult("SPRING")
    # Get a list of all springs
    springs = Model.Connections.GetChildren(DataModelObjectCategory.Spring, False)

    for spring in springs:
        print(spring.Name)

        spring_data = solver_data.GetObjectData(spring)
        element_id = spring_data.ElementId

        fForce = spring_results.GetElementValues(element_id)
        print(fForce[0])
```

Export a Result Object to an STL File

Goal: Export a result object to an STL file.

Code:

```
result = Model.Analyses[0].Solution.Children[1]
result.ExportToSTLFile("E:\\test.stl")
```

Export Result Images to Files

Goal: Export all results in the tree to PNG (2D image) and AVZ (3D image) files.

Code:

```
# get a list of all the results in the project
results = DataModel.GetObjectsByType(DataModelObjectCategory.Result)

#loop over the results
for result in results:

    # select and activate the result
    result.Activate()

    # export the result to avz file using the result name for the filename
    avzFilename = "D:\\Images\\" + result.Name + ".avz"
    Graphics.Export3D(avzFileName)

    # export the result as a 2D PNG file
    Graphics.ExportImage("D:\\images\\" + result.Name + ".png")

print "Done with Exporting Results"
```

Tag and Group Result Objects Based on Scoping and Load Steps

Goal:

- Add results based on the number of load steps
- Assign tags to created results based on scoping and type of result
- Group results based on tags

Code:

```
# Get the number of steps for the analysis
analysis_settings = Model.Analyses[0].AnalysisSettings
number_of_steps = analysis_settings.NumberOfSteps

# Get the Named Selection of interest
solution = Model.Analyses[0].Solution
bolt = DataModel.GetObjectsByName("Bolt")[0] #This is a named selection!!

with Transaction():
    # Create tags that will be used later for finding objects
    tag2 = Ansys.Mechanical.Application.ObjectTag("Bolt")
    tag3 = Ansys.Mechanical.Application.ObjectTag("U Sum")
    tag4 = Ansys.Mechanical.Application.ObjectTag("EQV")
    DataModel.ObjectTags.Add(tag2)
    DataModel.ObjectTags.Add(tag3)
    DataModel.ObjectTags.Add(tag4)

    # For each step add the desired result objects with appropriate settings
    for step in range(1, number_of_steps):
        u_result = solution.AddTotalDeformation()
        u_result.Name = "Total Deformation @ " + str(step) + " sec"
        u_result.DisplayTime = analysis_settings.GetStepEndTime(step)
        # Apply tags
        tag3.AddObject(u_result)

        s_result = solution.AddEquivalentStress()
        s_result.Name = "Eqv Stress @ " + str(step) + " sec"
        tag4.AddObject(s_result)

    for step in range(1, number_of_steps):
        u_result = solution.AddTotalDeformation()
        u_result.Name = "Bolt Deformation @ " + str(step) + " sec"
        u_result.Location = bolt
        u_result.DisplayTime = analysis_settings.GetStepEndTime(step)
```

```
        u_result.CalculateTimeHistory = False
        tag2.AddObject(u_result)
        tag3.AddObject(u_result)

        s_result = solution.AddEquivalentStress()
        s_result.Name = "Bolt Stress @ " + str(step) + " sec"
        s_result.Location = bolt
        tag2.AddObject(s_result)
        tag4.AddObject(s_result)

    tag2_list = tag2.Objects
    tag3_list = tag3.Objects
    tag4_list = tag4.Objects

    # Find similar objects using the tags
    u_all = [x for x in tag3_list if x not in tag2_list]
    u_bolt = [x for x in tag3_list if x in tag2_list]
    s_all = [x for x in tag4_list if x not in tag2_list]
    s_Bolt = [x for x in tag4_list if x in tag2_list]

# Group similar objects
group = Tree.Group(u_all)
group.Name = "Total Deformation (All Bodies)"
group = Tree.Group(u_bolt)
group.Name = "Total Deformation (Bolt)"
group = Tree.Group(s_all)
group.Name = "Stress (All Bodies)"
group = Tree.Group(s_Bolt)
group.Name = "Stress (Bolt)"

Tree.Activate([solution])
```

Work with Solution Combinations

Goal: Create a solution combination object combining many environments.

Code:

```
# get the environments, an alternative would be via Model.Analyses
envs = DataModel.GetObjectsByType(DataModelObjectCategory.Analysis)

# create a solution combination object; By default it will come with 1 base case and 1 combination
sc = Model.AddSolutionCombination()

# definition object holds all the data
scdef = sc.Definition

# Any property on a base case can be set using an index and value
scdef.SetBaseCaseAnalysis(0, envs[0])
scdef.SetBaseCaseTime(0,1)

# Add more base cases as you desire
scdef.AddBaseCase()
scdef.SetBaseCaseAnalysis(1, envs[1])

# You can even pass in the Base Case settings to the constructor (Arguments are name, analysis, time)
scdef.AddBaseCase("BC 3", envs[1], 2)

# Any property on a load combination can be set using an index and value
scdef.SetLoadCombinationType(0,1)

# Add more load combinations as you desire
scdef.AddLoadCombination()
scdef.SetLoadCombinationName(1, "LC2")

# You can even pass in the Load Combination settings to the constructor (Arguments are name, type)
scdef.AddLoadCombination("LC3", 1)
```

```
# Coefficients are set using two indices and a value
scdef.SetCoefficient(0, 0, 2)
scdef.SetCoefficient(0, 2, 1)
scdef.SetCoefficient(1, 0, -0.5)
scdef.SetCoefficient(1, 1, 0.75)
scdef.SetCoefficient(2, 0, -1)
scdef.SetCoefficient(2, 2, 1.5)

#Once fully defined, add results and evaluate
sc.AddEquivalentStress()
sc.EvaluateAllResults()
```

Create a Pressure Load

Goal: Create a pressure on the first face of the first body for the first part.

Code:

```
#The following example creates a pressure on the first face of the first body for the first part.
pressure = Model.Analyses[0].AddPressure()
part = Model.Geometry.Children
body1 = part1.Children[0]
face1 = body1.GetGeoBody().Faces[0]  # Get the first face of the body.
selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
selection.Entities = [face1]
pressure.Location = selection
pressure.Magnitude.Inputs[0].DiscreteValues = [Quantity("0 [s]"), Quantity("1 [s]")]
pressure.Magnitude.Output.DiscreteValues = [Quantity("10 [Pa]"), Quantity("20 [Pa]")]
```

Create Node Merge Object at a Symmetry Plane

Goal: Create a node merge object on a model with a symmetry plane.

Code:

```
# Purpose of the script: create a node merge object on models with a (plane) symmetry
# How to use: select a coordinate that defines a symmetry plane (origin axisX, axisY)
#             then the script will create a node merge object with faces automatically selected

def GetSignedDistanceFromPointToPlane(point, planeOrigin, planeNormal):
    OP = [ planeOrigin[0]-point[0], planeOrigin[1]-point[1], planeOrigin[2]-point[2] ]
    cosOpNormal = OP[0]*planeNormal[0]+OP[1]*planeNormal[1]+OP[2]*planeNormal[2]
    normalNorm = abs(planeNormal[0]*planeNormal[0]+planeNormal[1]*planeNormal[1]+planeNormal[2]*planeNormal[2])
    dist = cosOpNormal / normalNorm
    return dist

def GetSelectedCoordinateSystem():
    if Tree.ActiveObjects.Count != 1
        return None
    obj = Tree.FirstActiveObject
    if not obj.Path.StartsWith('/Project/Model/Coordinate Systems'):
        return None
    return obj

def FindFacesClosestToPlane(planeOrigin, planeNormal, maximumDistanceToPlane):
    assembly = DataModel.GeoData.Assemblies[0]
    parts = assembly.Parts

    facesPlus = []
    facesMinus = []
    for part in parts:
        for body in part.Bodies:
            distBody = GetSignedDistanceFromPointToPlane(body.Centroid, planeOrigin, planeNormal)
            for face in body.Faces:
```

```
                distFace = GetSignedDistanceFromPointToPlane(face.Centroid, planeOrigin, planeNormal)
                if abs(distFace) <= maximumDistanceToPlane:
                    if distBody >= 0:
                        facesPlus.append(face)
                    else:
                        facesMinus.append(face)
    return (facesPlus, facesMinus)

def AddNodeMergeObject():
    meshEdits = DataModel.GetObjectsByType(DataModelObjectCategory.MeshEdit)
    if meshEdits.Count > 0:
        meshEdit = meshEdits[0]
    else:
        meshEdit = Model.AddMeshEdit()

    meshEdit.AddNodeMerge()
    nodeMergeObj = DataModel.GetObjectsByType(DataModelObjectCategory.NodeMerge)[0]
    return nodeMergeObj

def GetGeometryBoundingBoxLength():
    geom = Model.Geometry
    lengthQuantity = geom.LengthX*geom.LengthX + geom.LengthY*geom.LengthY + geom.LengthZ*geom.LengthZ
    return lengthQuantity.Value

def ShowError(errString):
    ExtAPI.Application.ScriptByName("jscript").ExecuteCommand("WBScript.Out('" + errString + "', 1)")

def CreateNodeMergeAtPlane():
    try:
        csObj = GetSelectedCoordinateSystem()
        if csObj is None:
            raise Exception("Select a coordinate system that defines the symmetry plane as (origin, AxiX, AxisY)")

        planeOrigin = csObj.Origin
        planeNormal = csObj.ZAxis
        maximumDistanceToPlane = GetGeometryBoundingBoxLength() * 1e-5
        facesColls = FindFacesClosestToPlane(planeOrigin, planeNormal, maximumDistanceToPlane)

        nodeMergeObj = AddNodeMergeObject()

        masterSel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
        masterSel.Entities = facesColls[0]
        slaveSel = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
        slaveSel.Entities = facesColls[1]

        nodeMergeObj.MasterLocation = masterSel
        nodeMergeObj.SlaveLocation = slaveSel
    except Exception as ex:
        ShowError("Error: {0}".format(ex))
```

Access Contour Results for an Evaluated Result

Goal: For a solved result, access the nodal/elemental values and unit label.

Code:

```
Model=ExtAPI.DataModel.Project.Model
#select the result object on the tree
result=Tree.FirstActiveObject
#First result item can also be accessed with
result=Model.Analyses[0].Solution.Children[1]

#Plot the nodal/elemental values using PlotData
print("The values for " + result.Name + " is")
result.PlotData
#For accessing result, with column name (“Values”)
result.PlotData["Values"]
#For accessing values on the 5th row
```

```
result.PlotData["Values"][4] #Array starts at 0
#For accessing result value components
result.PlotData.Dependents
#For accessing node/element IDs
result.PlotData.Independents
#To get the unit label for the values
result.PlotData["Values"].Unit
```

Write Contour Results to a Text File

Goal: For a solved result, write the nodal/elemental values to a text file using **`PlotData`**.

Code:

```
Model=ExtAPI.DataModel.Project.Model
#select the result object on the tree
result=Tree.FirstActiveObject
resultValues= result.PlotData["Values"]
nodeList=result.PlotData["Node"]
with open('C:\Users\Admin\Desktop\Testfile.txt','w') as testfile:
    for ii in range(len(nodeList)):
        a=nodeList[ii]
        b=resultValues[ii]
        wrt=str(ii)+'\t'+str(a)+"\t"+str(b)+"\n"
        testfile.write(wrt)
```

Access Contour Results at Individual Nodes/Elements

Goal: For a solved result, get the result value at an individual node using **`PlotData`**.

Code:

```
Model=ExtAPI.DataModel.Project.Model

#select the result object on the tree
result=Tree.FirstActiveObject
plotDataResult= result.PlotData

#For node number = nodeID
def findNodeResult(nodeID,plotDataResult):
    nodes=plotDataResult ['Node']
    resultValue=plotDataResult ['Values']
    if nodeID in nodes:

        for index in range(len(nodes)):

            if nodes[index]==nodeID:
                return resultValue[index]
    else:
        print("Given NodeID doesn't exist in result set")

#Find the result value associated for node number 3 using

findNodeResult(3,plotDataResult)
```

Coordinate System Math

Goal: Transform the given global coordinates to the first non-global coordinate system object in the tree.

Code:

```
import units
import math

def MatrixTransformationGlobalToUserCS(global_coordinates, destination_coordinate_system, length_unit):
    from_unit = DataModel.CurrentConsistentUnitFromQuantityName("Length")
    factor = units.ConvertUnit(1, from_unit, length_unit, "Length")

    origin = destination_coordinate_system.Origin

    x_axis = destination_coordinate_system.XAxis
    y_axis = destination_coordinate_system.YAxis
    z_axis = destination_coordinate_system.ZAxis

    user_coordinates = []
    user_coordinates.append(x_axis[0]*(global_coordinates[0] - factor*origin[0]) +
                            x_axis[1]*(global_coordinates[1] - factor*origin[1]) +
                            x_axis[2]*(global_coordinates[2] - factor*origin[2]))
    user_coordinates.append(y_axis[0]*(global_coordinates[0] - factor*origin[0]) +
                            y_axis[1]*(global_coordinates[1] - factor*origin[1]) +
                            y_axis[2]*(global_coordinates[2] - factor*origin[2]))
    user_coordinates.append(z_axis[0]*(global_coordinates[0] - factor*origin[0]) +
                            z_axis[1]*(global_coordinates[1] - factor*origin[1]) +
                            z_axis[2]*(global_coordinates[2] - factor*origin[2]))

    if destination_coordinate_system.CoordinateSystemType == CoordinateSystemTypeEnum.Cartesian:
        return user_coordinates
    elif destination_coordinate_system.CoordinateSystemType == CoordinateSystemTypeEnum.Cylindrical:
        r = sqrt(user_coordinates[0] * user_coordinates[0] + user_coordinates[1] * user_coordinates[1])
        theta = math.degrees(math.atan(user_coordinates[1] / user_coordinates[0]))
        z = user_coordinates[2]
        return [r, theta, z]

def MatrixTransformationGlobalToUserCSUsingMatrix4D(global_coordinates, destination_coordinate_system, length_unit
    from_unit = ExtAPI.DataModel.CurrentConsistentUnitFromQuantityName("Length")
    factor = units.ConvertUnit(1, from_unit, length_unit, "Length")

    origin = destination_coordinate_system.Origin

    x_axis = destination_coordinate_system.XAxis
    y_axis = destination_coordinate_system.YAxis
    z_axis = destination_coordinate_system.ZAxis

    x_axis_vector = Vector3D(x_axis[0], x_axis[1], x_axis[2])
    y_axis_vector = Vector3D(y_axis[0], y_axis[1], y_axis[2])
    z_axis_vector = Vector3D(z_axis[0], z_axis[1], z_axis[2])

    identity  =  Matrix4D()
    transformation = identity.CreateSystem(x_axis_vector,y_axis_vector,z_axis_vector)

    vector = Vector3D(global_coordinates[0] - origin[0] * factor, global_coordinates[1] - origin[1] * factor, glob
    transformation.Transpose()
    vector_trans = transformation.Transform(vector)
    return vector_trans


global_coordinates = [20,40,60]
localCS = Model.CoordinateSystems.Children[1]

localCoordinates1 = MatrixTransformationGlobalToUserCS(global_coordinates, localCS, "mm")
localCoordinates2 = MatrixTransformationGlobalToUserCSUsingMatrix4D(global_coordinates, localCS, "mm")

print(localCoordinates1)
print(localCoordinates2)
```

Script Examples for Interacting with the Mechanical Session

The following scripts are for interacting with the Mechanical session:

Remesh a Model Multiple Times and Track Metrics

Goal: Remesh a model five times, tracking how long each remesh takes and the number of nodes that are created.

Code:

```
from time import clock, time
for x in range(0, 5):
    Model.ClearGeneratedData()
    t1 = time()
    Model.Mesh.GenerateMesh()
    t2 = time()
    print "Stats for mesh %d, elapsed time=%d" % (x+1, t2-t1)
    print Model.Mesh.Nodes
```

Scan Results, Suppress Any with Invalid Display Times, and Evaluate

Goal: Scan all result objects in your first analysis, suppress any results with invalid display times, and then evaluate all results.

Code:

```
aset = Model.Analyses[0].AnalysisSettings
OrigStep = aset.CurrentStepNumber
aset.CurrentStepNumber = aset.NumberOfSteps
FinalTime = aset.StepEndTime
aset.CurrentStepNumber = OrigStep
sol = Model.Analyses[0].Solution
for obj in sol.Children:
    if hasattr(obj,"DisplayTime"):
        if obj.DisplayTime > FinalTime:
```

```
            obj.Suppressed = True
sol.EvaluateAllResults()
```

Check Version

Goal: Check the version in which your user is operating.

Code:

```
import Ansys.Utilities
from Ansys.Utilities import ApplicationConfiguration
---
ansysVersion = ApplicationConfiguration.DefaultConfiguration.VersionInfo.VersionString
```

Check Operating Environment

Goal: Check to see if your user is operating in a Unix environment.

Code:

```
import Ansys.Utilities
from Ansys.Utilities import ApplicationConfiguration
---
isUnix = ApplicationConfiguration.DefaultConfiguration.IsUnix
```

Retrieve Stress Results

Goal: Obtain stress results for an element.

Code:

```
reader = Model.Analyses[0].GetResultsData()
reader.CurrentResultSet=1
S=reader.GetResult("S")
S.GetElementValues(1)
reader.Dispose()
```

Search for Keyword and Export

Goal: Loop through all results that contain a keyword of **`For Image`** and export the image shown to a directory.

You must specify the path in the first line.

Code:

```
path = "C:\\"
n = 0

for analysis in Model.Analyses:
    sol = analysis.Solution
    for sol_obj in sol.Children:
        if sol_obj.Name.Contains("For_Image"):
            n += 1
```

```
            sol_obj.Activate()
            Graphics.ExportImage(path + sol_obj.name + ".png")
```

Modify Export Setting

Goal: Modify the export setting to include node numbers in exports.

Code:

```
ExtAPI.Application.ScriptByName("jscript").ExecuteCommand('WB.PreferenceMgr.Preference("PID_Show_Node_Numbers") =
```

Pan the Camera

Goal: Pan the camera.

Code:

```
camera = Graphics.Camera
up_vector = camera.UpVector
view_vector = camera.ViewVector

#get the 2D CSYS in the screen plane based on the computed UpVector, derived from the prescribed one.
plane_right = up_vector.CrossProduct(view_vector)                       # the “x” axis of the 2D CSYS.
plane_up = up_vector - (up_vector.DotProduct(view_vector)) * view_vector # the “y” axis of the 2D CSYS.

#construct the pan vector in the screen plane (Use the units from Graphics.Unit).
pan_right = 100
pan_up = 100
pan_vector = plane_right * pan_right + plane_up * pan_up

#set the new focal point by adding the pan vector to the original focal point.
pan_origin = camera.FocalPoint.Location
new_x, new_y, new_z = (pan_origin[0] + pan_vector[0], pan_origin[1] + pan_vector[1], pan_origin[2] + pan_vector[2]
camera.FocalPoint = Point([new_x, new_y, new_z], Graphics.Unit)
```

Functions to Draw

Goal: Add several example draw functions that can be used to draw entities in the Mechanical graphics window.

Code:

```
def DrawElements():
    elems = DataModel.MeshDataByName('Global').Elements
    elem = Graphics.Scene.Factory3D.CreateMesh(list(elems)[:5])
    elem.Color = 0xA00ABC

def DrawBody():
    body = DataModel.GeoData.Assemblies[0].Parts[0].Bodies[0]
    geo = Graphics.Scene.Factory3D.CreateGeometry(body)
    geo.Color = 0xAAA000

def DrawFace():
    face = DataModel.GeoData.Assemblies[0].Parts[0].Bodies[0].Faces[0]
    geo = Graphics.Scene.Factory3D.CreateGeometry(face)
    geo.Color = 0xABCABC

def DrawEdge():
    edge = DataModel.GeoData.Assemblies[0].Parts[0].Bodies[0].Faces[0].Edges[1]
```

```
    geo = Graphics.Scene.Factory3D.CreateGeometry(edge)
    geo.LineWeight = 14
    geo.VertexColor = 0x0000AA
    geo.VertexSize = 15
    geo.Color = 0x000ABC

def DrawVertex():
    vertex = DataModel.GeoData.Assemblies[0].Parts[0].Bodies[0].Faces[0].Edges[0].Vertices[0]
    geo = Graphics.Scene.Factory3D.CreateGeometry(vertex)
    geo.Color = 0xBACBAC

def DrawTriad():
    triad = Graphics.Scene.Factory3D.CreateTriad(1.0)

def DrawArrow():
    Vector3D = Graphics.CreateVector3D
    arrow = Graphics.Scene.Factory3D.CreateArrow(0.5)
    arrow.Color = 0xFFABC0
    arrow.Transformation3D.Translate(Vector3D(0.5, 0.5, 1))

def DrawBox():
    box = Graphics.Scene.Factory3D.CreateBox(2.0, 3.0, 4.0)
    box.Color = 0xFF0ABC

def DrawCircle():
    cone = Graphics.Scene.Factory3D.CreateCircle(0.5)
    cone.Color = 0xAAACCC

def DrawCone():
    cone = Graphics.Scene.Factory3D.CreateCone(3.0, 2.0, 1.0)
    cone.Color = 0xABC000

def DrawCylinder():
    cone = Graphics.Scene.Factory3D.CreateCylinder(0.5, 2)
    cone.Color = 0xAAA000

def DrawDisc():
    cone = Graphics.Scene.Factory3D.CreateDisc(0.5)
    cone.Color = 0xAAABBB

def DrawQuad():
    sphere = Graphics.Scene.Factory3D.CreateQuad(0.33, 0.33)
    sphere.Color = 0xFFA000

def DrawShell():
    shell = Graphics.Scene.Factory3D.CreateShell([1., 1., 1. ,2., 1., 1., 1., 1. ,2.], [0., 1., 0., 0., 1., 0., 0.
    shell.Color = 0xFFFFFF

def DrawSphere():
    sphere = Graphics.Scene.Factory3D.CreateSphere(0.33)
    sphere.Color = 0xFF0A00

def DrawPolyline3D():
    Point2D = Graphics.CreatePixelPoint
    Point3D = Graphics.CreateWorldPoint
    Vector3D = Graphics.CreateVector3D

    with Graphics.Suspend():
        p1 = Point2D(10, 10)
        p2 = Point2D(10, 100)
        p3 = Point2D(100, 100)
        p4 = Point2D(100, 10)

        coll = Graphics.Scene.CreateChildCollection()
        l1 = coll.Factory2D.CreatePolyline([p1, p2, p3, p4])
        l1.Closed = True

        p5 = Point2D(0, 0)
        p6 = Point2D(100, 100)
        l2 = Graphics.Scene.Factory2D.CreatePolyline([p5, p6])

        p7 = Point2D(20, 40)
```

```
        text = Graphics.Scene.Factory2D.CreateText(p7, "Hello World 3D")

        wp1 = Point3D(0, 5, 0)
        wp2 = Point3D(0, 0, 0)
        wp3 = Point3D(5, 0, 0)

        coll = Graphics.Scene.CreateChildCollection()

        l1 = coll.Factory3D.CreatePolyline([wp1, wp2])
        l2 = coll.Factory3D.CreatePolyline([wp2, wp3])

    #shell = Graphics.Scene.Factory3D.CreateShell([1.,1.,1.,2.,1.,1.,1.,1.,2.], [0.,1.,0.,0.,1.,0.,0.,1.,0.], [0,1

    points = []
    for i in range(10):
     for j in range(10):
      for k in range(10):
       points.Add(Point3D(float(i)/float(10), float(j)/float(10),float(k)/float(10)))

    coll.Factory3D.CreatePoint(points, 4)

    return True

    for i in range(5):
     for j in range(5):
      for k in range(5):
       point = Point3D(float(i)/float(2), float(j)/float(2),float(k)/float(2))
       coll.Factory2D.CreateText(point, str(i + j + k))

def DrawPolyline2D():
    Point2D = Graphics.CreatePixelPoint
    Point3D = Graphics.CreateWorldPoint

    with Graphics.Suspend():

        p1 = Point2D(10, 10)
        p2 = Point2D(10, 100)
        p3 = Point2D(100, 100)
        p4 = Point2D(100, 10)

        coll = Graphics.Scene.CreateChildCollection()
        l1 = coll.Factory2D.CreatePolyline([p1, p2, p3, p4])
        l1.Closed = True

        p5 = Point2D(0,0)
        p6 = Point2D(100,100)
        l2 = Graphics.Scene.Factory2D.CreatePolyline([p5, p6])

        p7 = Point2D(20,40)
        text = Graphics.Scene.Factory2D.CreateText(p7, "Hello World 2D")

        wp1 = Point3D(0,5,0)
        wp2 = Point3D(0,0,0)
        wp3 = Point3D(5,0,0)

        coll = Graphics.Scene.CreateChildCollection()

        l1 = coll.Factory3D.CreatePolyline([wp1, wp2])
        l2 = coll.Factory3D.CreatePolyline([wp2, wp3])

    points = []
    for i in range(10):
     for j in range(10):
       points.Add(Point2D(float(i)/float(10), float(j)/float(10)))

    coll.Factory2D.CreatePoint(points, 4)
```

Export All Result Animations

Goal: Export the animation of the results in the model to a WMV video file with given resolution, frames and duration.

Code:

```
#Set camera and view
cam = Graphics.Camera
cam.SetSpecificViewOrientation(ViewOrientationType.Right)

#Animation settings
Graphics.ResultAnimationOptions.NumberOfFrames = 10
Graphics.ResultAnimationOptions.Duration = Quantity(2, 's')
settings = Ansys.Mechanical.Graphics.AnimationExportSettings(width = 1000, height = 665)

for analysis in Model.Analyses:
 results = analysis.Solution.GetChildren(DataModelObjectCategory.Result, True)
 for result in results:
  location ="C:\\Samples\\WMV\\" + result.Name + ".wmv"
  result.ExportAnimation(location, GraphicsAnimationExportFormat.WMV, settings)
```