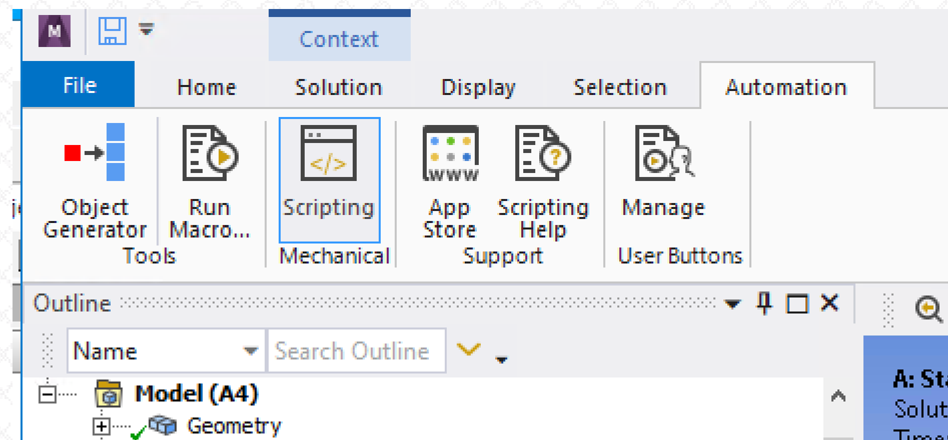
Using the ACT Console for Automation in Mechanical

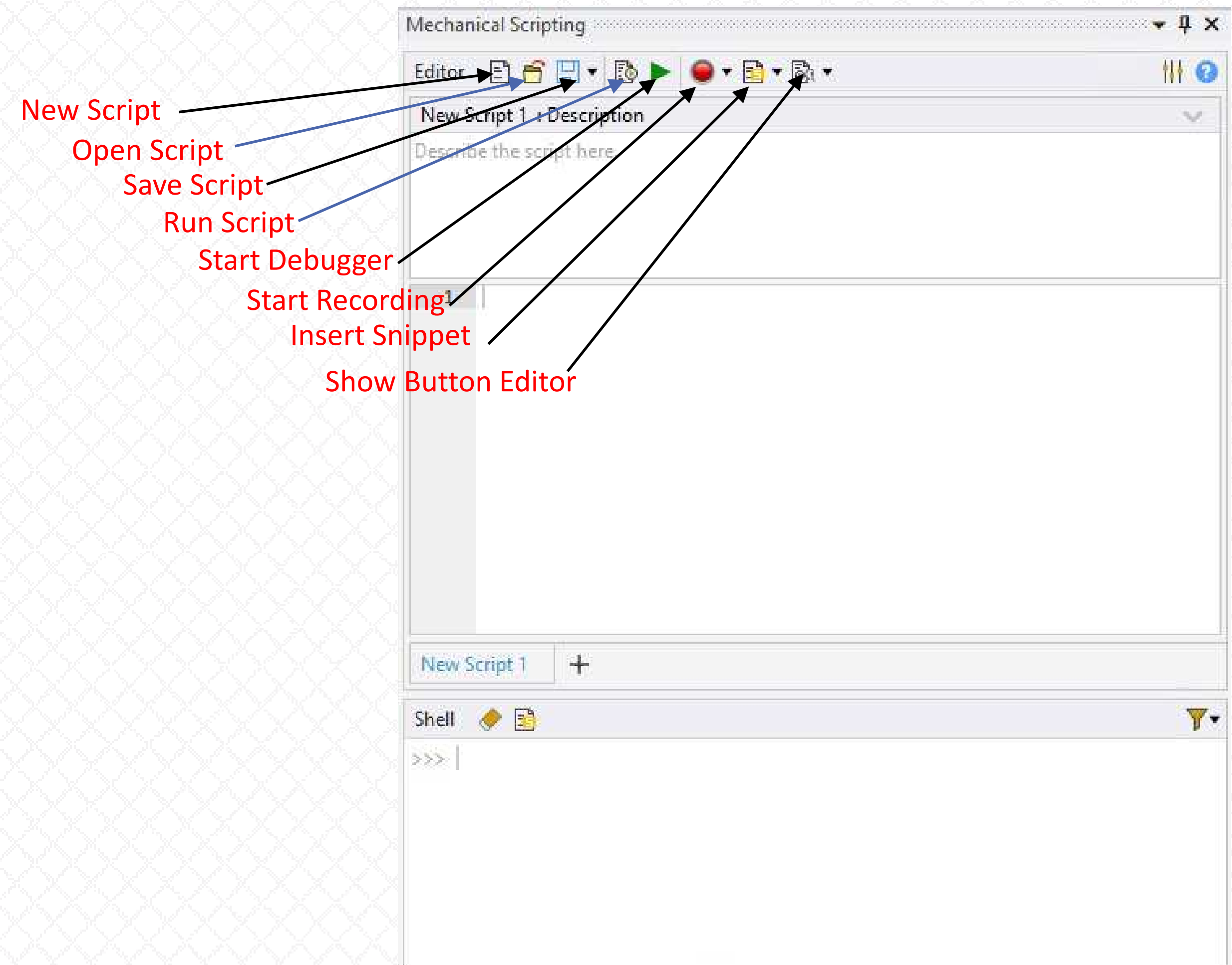
The ACT Console in Mechanical: Backgroung

- ACT (the Ansys Customization Toolkit) is the main (preferred) programming interface to the Ansys suite of products. It is built on top of an underlying .NET framework, and so inherits much of the functionality found there
- PADT has written some blog posts in the past which cover some of this functionality (one example, is the choice of language itself –see here)
- When discussing ACT, Ansys tends to make a distinction between "customization" and "process compression or automation"
- In the former, the user creates some new (custom) functionality using ACT –called an "extension". This typicallly involves the creation of a Graphical User Interface (GUI) in addition to the functional code
- in the latter, the user is simply creating a few of lines of code to execute when needed to speed up or formalize some laborious process
- Good tutorials and examples exist for creating ACT extensions (the customization aspect: see here for example) –as well an an extensive 'app' store for download here)
- In this article, we want to focus on just the automation aspect of ACT in Mechanical. In particular, we want to show how users can rather quickly and easily write a short script to a automate a procedure in Mechanical using the ACT console.

The Mechanical Scripting Editor (ACT Console)

- The ACT console in Mechanical has been around for some time now, and is evolving.
- Ansys refers to this console as the “Mechanical Scripting Editor”.
- As of version 2020R1, the Mechanical scripting editor is invoked from the ‘Automation’ tab in the top menu and supprts the following features





PADT
We Make Innovation Work
www.padtinc.com

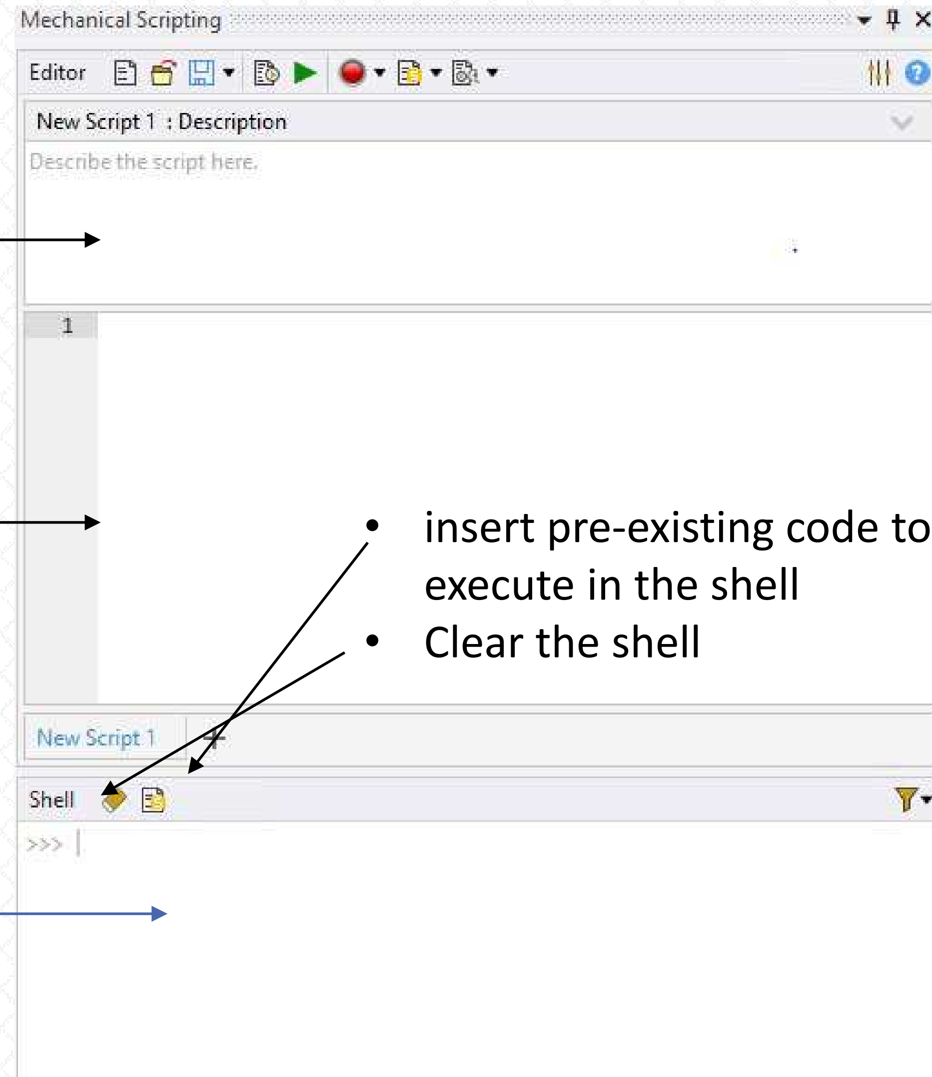
The Mechanical Scripting Editor (ACT Console)

- There are three window panes whose functionality is as described below

Description. A short description of the script goes here

The **editor**.The contents of the current script go here

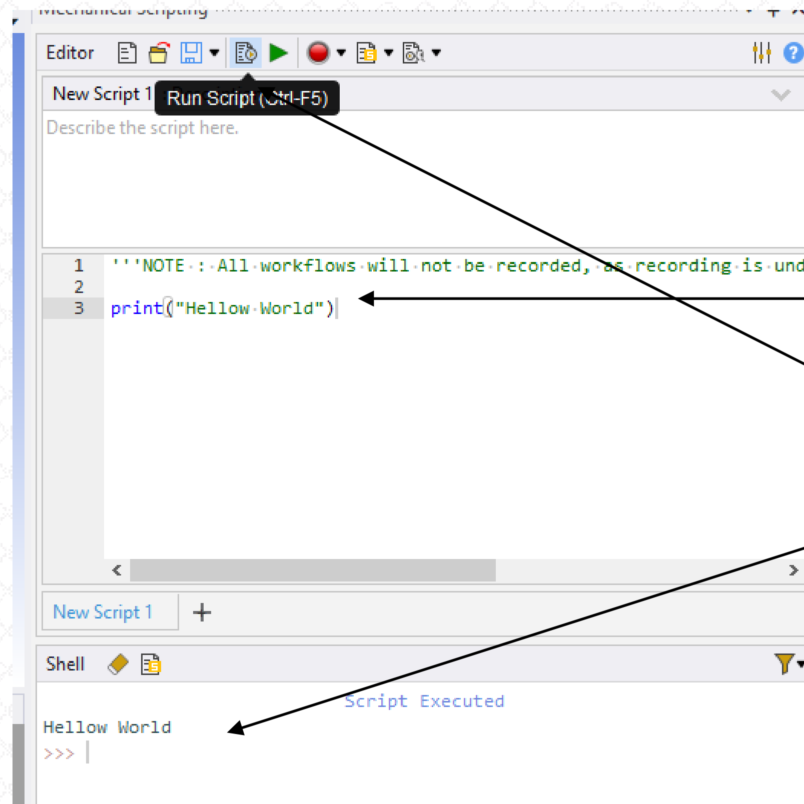
The console, or "**Shell**". This is where lines of code actually get executed



- insert pre-existing code to execute in the shell
- Clear the shell

The Mechanical Scripting Editor (ACT Console)

- The Mechanical Scripting Editor only interprets the python language
- As explained elsewhere (and in previous blog posts), ACT supports the IronPython version of the python programming language
- IronPython is the only .NET-supported language that the Mechanical Scripting Editor can interpret



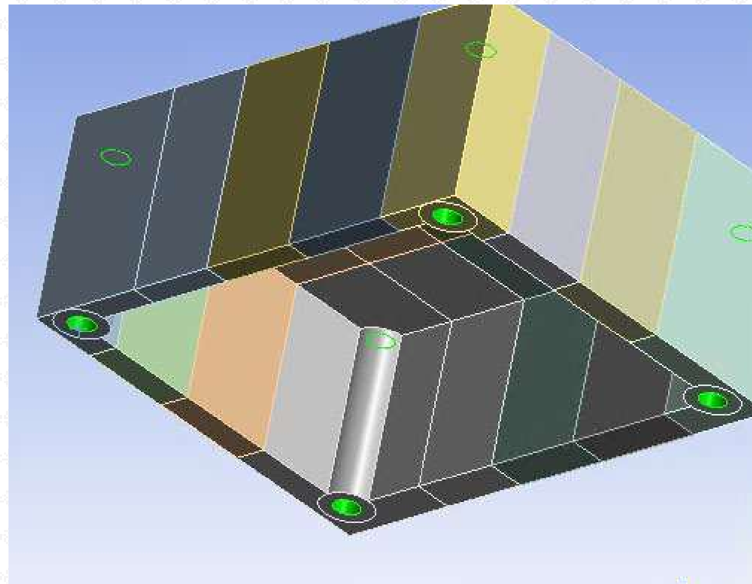
- Type this line in the script editor (the 'editor' window)
- Hit the 'run script' button to see results in the shell

The Mechanical Scripting Editor (ACT Console)

- All commands entered into the shell get executed within a certain namespace. Let’s call it the “entry point”.
- The following five Ansys objects are available in this entry point
    - **ExtAPI:** Short for “External Application Programming Interface”. This is the top-level for accessing all available variables and objects
    - **Tree:** Controls everything in the mechanical tree
    - **Model:** Allows one to browse the model data, mesh, geometry...
    - **DataModel:** Reads all that was created in a model
    - **Graphics:** Controls the graphic window, take screen shot, draw...
- These objects have a LOT of redundancy (there are often multiple ways to do something).
- For example, the following two lines achieve the same end (running the model):
    - DataModel.AnalysisList[0].Solution.Solve()
    - Model.Analyses[0].Solve()

The Mechanical Scripting Editor (ACT Console)

- A major enhancement to the Mechanical Scripting Editor at 2020R1 is the indtroduction of the ‘Start Recording’ button
- This allows the user to capture GUI operations in the Mechanical Scripting Editor*
- To see how this enhances productivity, let’s first use it to apply a fixed support to the four cylindrical blind hole features shown below (highlighted green)
- This model is supplied with this article as a 2020R1 archive

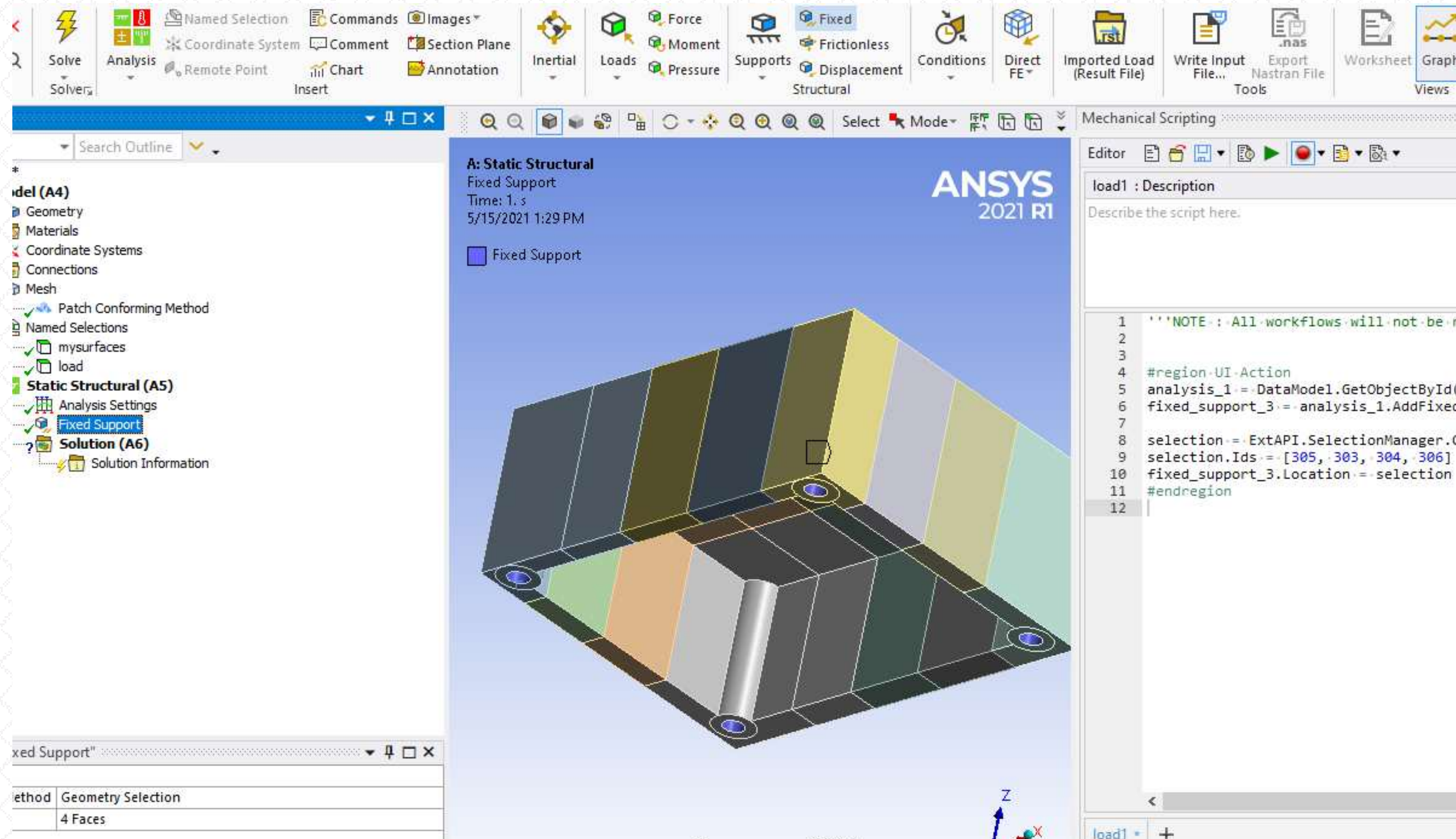


- Open the archive
- Launch the Scripting Editor as shown in slide 3

*This functionality is brand new at 2020R1. Not all operations are recorded. Future versions will support more operations. This is a a work in progress!

The Mechanical Scripting Editor (ACT Console)

- Make sure all panes are clear
- Then hit the “Start Recording” button
- In the GUI, select the four holes, and then in top tab, select ‘Environment->Fixed’



Note:
The fixed support is created, AND the associated ACT code automatically populates the editor!

The Mechanical Scripting Editor (ACT Console)

- It's worth taking a critical look at the code that's generated

```
#region UI Action
analysis_1 = DataModel.GetObjectById(61)
fixed_support_2 = analysis_1.AddFixedSupport()

selection =
ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
selection.Ids = [303, 305, 306, 304]
fixed_support_2.Location = selection
#endregion
```

- auto-generated comment: The start of a single recorded action
- Get the analysis object (the environment – 'Static Structural')
- Add fixed support by invoking the 'AddFixedSuport' method of the analysis object
- Get the four selected hole surface ID's and place in a 'SelectionInfo' object
- Point the fixed support object to the contents of the SelectionInfo object
- auto-generated comment: End of single recorded action

Note:

- The recorder seems to identify all objects by their DataModel ID's
- If you want the script to be more robust (what if the model changes?), you will want to minimize the number of objects you reference thsi way. For example, we can replace the first three lines with this one...

fixed_support_2= = Model.Analyses[0].AddFixedSupport()

The Mechanical Scripting Editor (ACT Console)

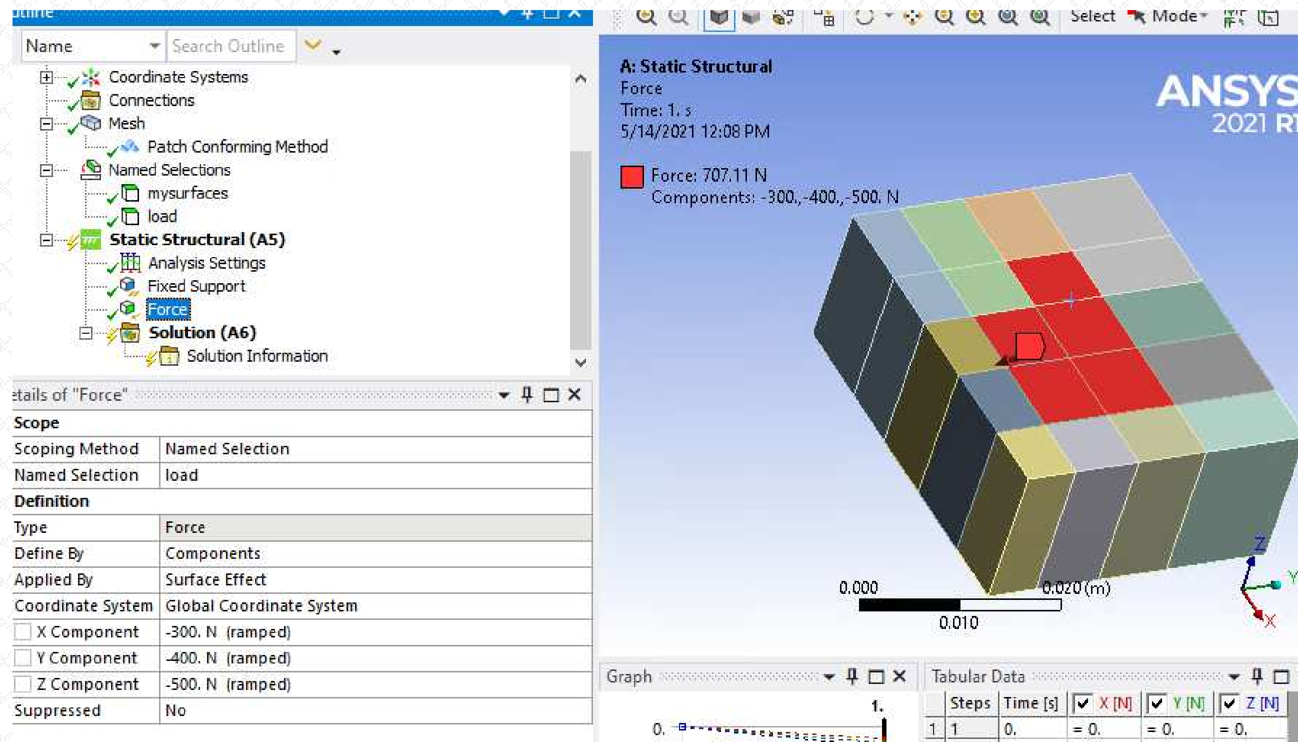
- Another Note: numbering found in the code reflects the sequence of variable creation by the recorder. I'm generating a 'fixed_support_2' here because I deleted a 'fixed_suport_1' previously
- Details like this can make for confusing code. It's therefore a good idea to rename variables to make more sense for future applications
- The takeway here is to use the recorder feature only as an aid –a shortcut to looking things up.
- But always clean up later...

- Delete the fixed support from the model in Workbench
- Stop the recorder
- Clear the editor and cut-and-paste the following cleaned up code into the editor pane (or open the 'fixedsupport.py' script from the user_files folder)
- Hit "Run Script"

```
fixed_support = Model.Analyses[0].AddFixedSupport()
selection = 
ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEnti
ties)
selection.Ids = [303, 305, 306, 304]
fixed_support.Location = selection
```

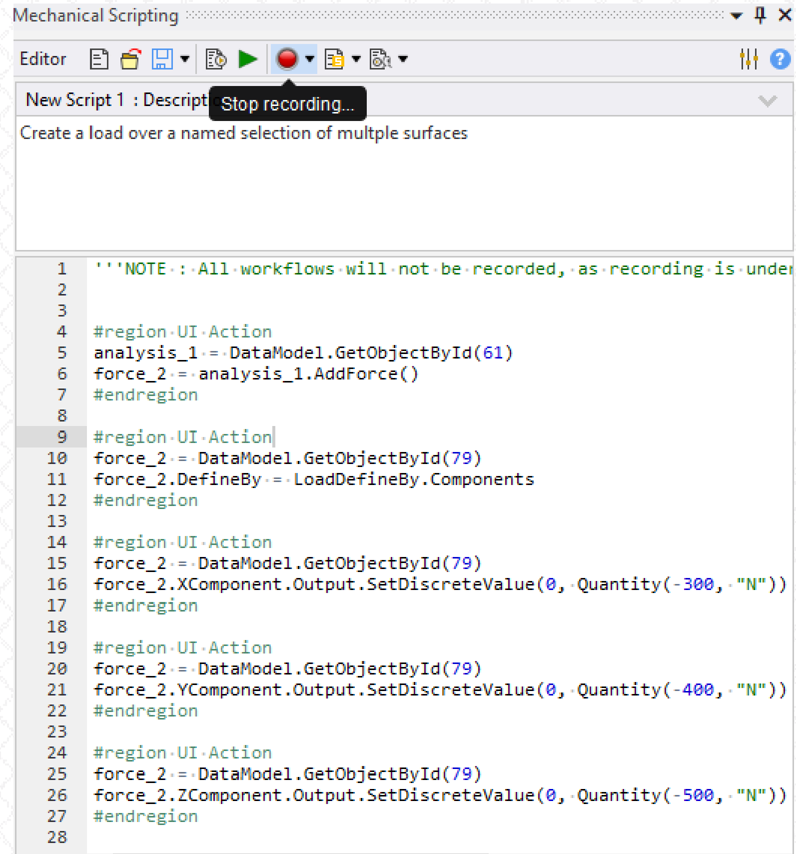
The Mechanical Scripting Editor (ACT Console)

- Ok. Now, after clearing the editor, start the recorder again and record the creation of the load shown below
- The load is applied on the surfaces shown and has the components {FX,FY,FZ} = {-300N, -400N, -500N}



The Mechanical Scripting Editor (ACT Console)

- The preceding generates the following lines of code
- turn the recording button ‘off’ at this point



The Mechanical Scripting Editor (ACT Console)

- To test this code, we should just need to delete the ‘Force’ object we created (and recorded) in the tree, and run the script...
- However, when we do so, we get the ‘run-time’ error shown below

mysurfaces
load
Static Structural (A5)
Analysis Settings
Force
Solution (A6)
Solution Information

etails of "Force"

Scope	
Scoping Method	Geometry Selection
Geometry	5 Faces
Definition	
Type	Force
Define By	Vector
Applied By	Surface Effect
Magnitude	0. N (ramped)
Direction	Click to Define
Suppressed	No

```
3   #region UI Action
4   analysis_1 = DataModel.GetObjectById(61)
5   force_1 = analysis_1.AddForce()
6
7   selection = ExtAPI.SelectionManager.CreateSelectionInfo(Selectic
8   selection.Ids = [167, 111, 120, 176, 261]
9   force_1.Location = selection
10  #endregion
11
12  #region UI Action
13  force_1 = DataModel.GetObjectById(552)
14  force_1.DefineBy = LoadDefineBy.Components
15  #endregion
16
17  #region UI Action
18  force_1 = DataModel.GetObjectById(552)
19  force_1.XComponent.Output.SetDiscreteValue(0, Quantity(-300, "N'
20  #endregion
21
22  #region UI Action
23  force_1 = DataModel.GetObjectById(552)
24  force_1.YComponent.Output.SetDiscreteValue(0, Quantity(-400, "N'
25  #endregion
26
27  #region UI Action
28  force_1 = DataModel.GetObjectById(552)
29  force_1.ZComponent.Output.SetDiscreteValue(0, Quantity(-500, "N'
30  #endregion
31
32
```

fixedsuppor... * fixedsupport load1 * STLExport * +

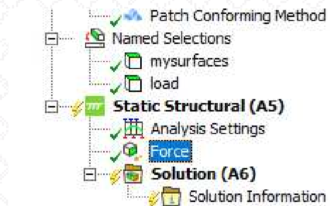
Shell

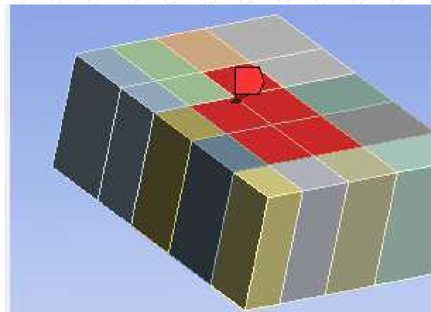
```
Script Executed
'NoneType' object has no attribute 'DefineBy'
```

- This is happening because the recorder references the old force object by its datamodel ID (which is no longer valid)

The Mechanical Scripting Editor (ACT Console)

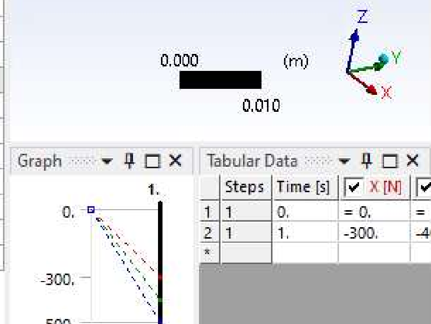
- So, let's test the code again.
- We'll delete the force object from the tree (again)
- This time, make sure to comment out all (redundant) ID references to the load we're creating





Details of "Force"

Scope	
Scoping Method	Geometry Selection
Geometry	5 Faces
Definition	
Type	Force
Define By	Components
Applied By	Surface Effect
Coordinate System	Global Coordinate System
X Component	-300. N (ramped)
Y Component	-400. N (ramped)
Z Component	-500. N (ramped)
Suppressed	No



```
1
2
3  #region UI Action
4  analysis_1 = DataModel.GetObjectById(61)
5  force_1 = analysis_1.AddForce()
6
7  selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
8  selection.Ids = [167, 111, 120, 176, 261]
9  force_1.Location = selection
10 #endregion
11
12 #region UI Action
13 #force_1 = DataModel.GetObjectById(552)
14 force_1.DefineBy = LoadDefineBy.Components
15 #endregion
16
17 #region UI Action
18 #force_1 = DataModel.GetObjectById(552)
19 force_1.XComponent.Output.SetDiscreteValue(0, Quantity(-300, "N"))
20 #endregion
21
22 #region UI Action
23 #force_1 = DataModel.GetObjectById(552)
24 force_1.YComponent.Output.SetDiscreteValue(0, Quantity(-400, "N"))
25 #endregion
26
27 #region UI Action
28 #force_1 = DataModel.GetObjectById(552)
29 force_1.ZComponent.Output.SetDiscreteValue(0, Quantity(-500, "N"))
30 #endregion
31
32
```

fixedsuppor... * fixedsupport load1 * STLExport * +

Shell

Script Executed

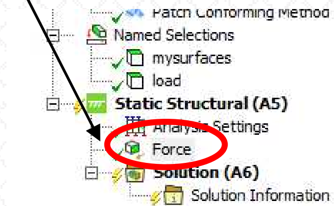
>>>

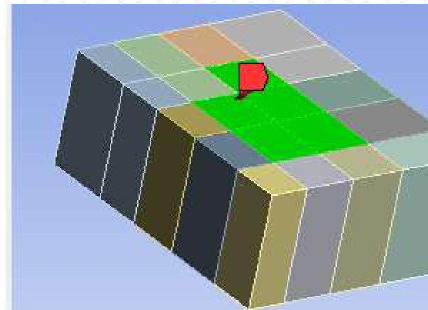
- Comment out ID selections of the new force object

We Make Innovation Work
www.padtinc.com

The Mechanical Scripting Editor (ACT Console)

- Hit “Run Script” again
- This time it works !





Details of "Force"

Scope	
Scoping Method	Geometry Selection
Geometry	5 Faces
Definition	
Type	Force
Define By	Components
Applied By	Surface Effect
Coordinate System	Global Coordinate System
X Component	-300. N (ramped)
Y Component	-400. N (ramped)
Z Component	-500. N (ramped)
Suppressed	No

0.000 (m) 0.010

Graph

Tabular Data

	Steps	Time [s]	X [N]	
1	1	0.	= 0.	=
2	1	1.	-300.	-4
*				

```
1
2
3  #region UI Action
4  analysis_1 = DataModel.GetObjectById(61)
5  force_1 = analysis_1.AddForce()
6
7  selection = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum
8  selection.Ids = [167, 111, 120, 176, 261]
9  force_1.Location = selection
10 #endregion
11
12 #region UI Action
13 #force_1 = DataModel.GetObjectById(552)
14 force_1.DefineBy = LoadDefineBy.Components
15 #endregion
16
17 #region UI Action
18 #force_1 = DataModel.GetObjectById(552)
19 force_1.XComponent.Output.SetDiscreteValue(0, Quantity(-300, "N"))
20 #endregion
21
22 #region UI Action
23 #force_1 = DataModel.GetObjectById(552)
24 force_1.YComponent.Output.SetDiscreteValue(0, Quantity(-400, "N"))
25 #endregion
26
27 #region UI Action
28 #force_1 = DataModel.GetObjectById(552)
29 force_1.ZComponent.Output.SetDiscreteValue(0, Quantity(-500, "N"))
30 #endregion
31
32
```

fixedsuppor... * fixedsupport load1 * STLExport * +

Shell

- Notice all redundant ID references to force object commented

The Mechanical Scripting Editor (ACT Console)

- This experience reminds us that the ‘record’ feature is still in its infancy
- Among other things, it makes redundant reference to object ID’s that get created in the recording
- This last feature requires us to comment or remove code that will become broken upon re-use (created object ID’s will change the next time we re-run the code)

- We can make the code still more robust by removing ALL object ID references (not just of objects our script creates) AND by adding code that selects our named selection for us...
- Clear the editor (by deleting all highlighted text in the editor) and the shell (hit “Clear Contents”), and delete the force object again
- Cut-and-paste following block of code into th editor and hit “Run Script” (or “Open script” “load1.py” from the user_files folder)

```
mycomp = Tree.Find(name="load")[0].Location
selection = ExtAPI.SelectionManager.NewSelection(mycomp)
force=Model.Analyses[0].AddForce()
force.DefineBy = LoadDefineBy.Components

force.XComponent.Output.SetDiscreteValue(0, Quantity(-300, "N"))
force.YComponent.Output.SetDiscreteValue(0, Quantity(-400, "N"))
force.ZComponent.Output.SetDiscreteValue(0, Quantity(-500, "N"))
```

‘Find’ the named selection in the tree (instead of manually selecting it)

‘Select’ the named selection

Don’t reference any object ID’s

Removed all redundant object references

The Mechanical Scripting Editor (ACT Console)

- This time, we didn't even have to reference the geometric entities making up our named selection (our code found the associated named selection)!
- This is now fairly robust. In fact, you can keep hitting "Run Script" and it will keep adding the same load

load1.py : Description

Create a load over a named selection of multple surfaces

```
mycomp = Tree.Find(name="load")[0].Location
selection = ExtAPI.SelectionManager.NewSelection(mycomp)
force=Model.Analyses[0].AddForce()
force.DefineBy = LoadDefineBy.Components

force.XComponent.Output.SetDiscreteValue(0, Quantity(-300, "N"))
force.YComponent.Output.SetDiscreteValue(0, Quantity(-400, "N"))
force.ZComponent.Output.SetDiscreteValue(0, Quantity(-500, "N"))
```

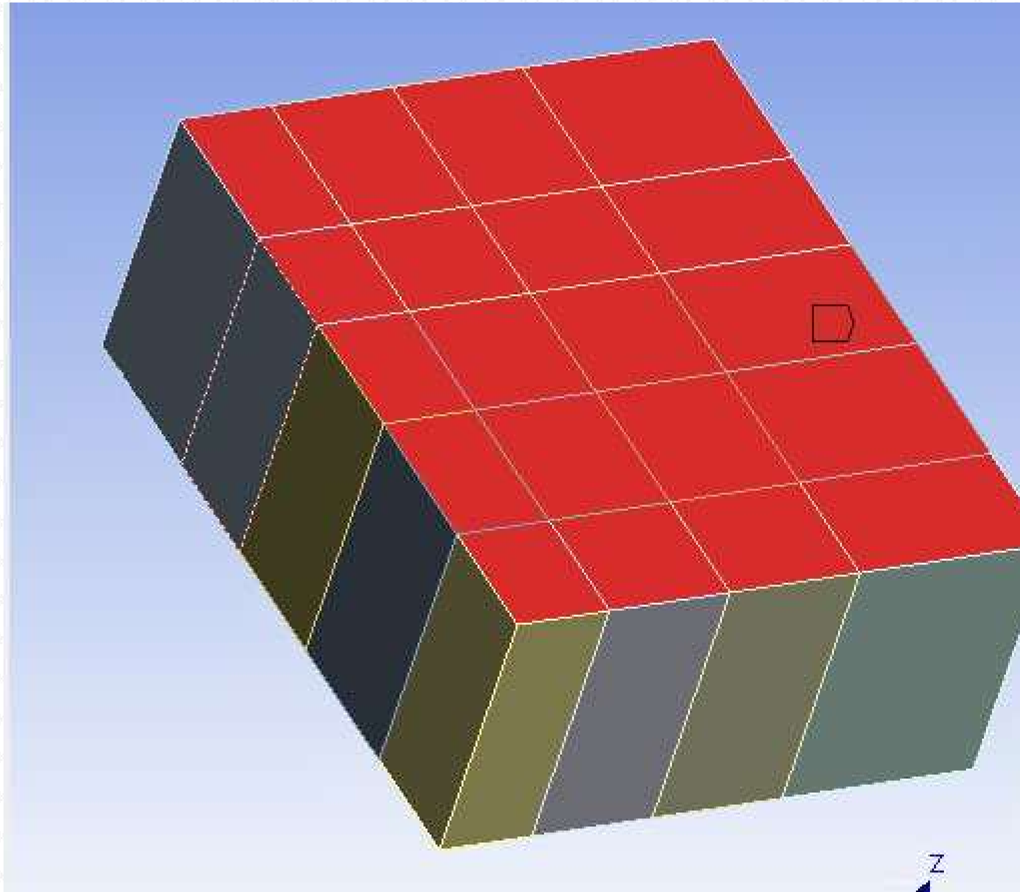
A: Static Structural
Force
Time: 1. s
5/14/2021 5:29 PM
Force: 707.11 N
Components: -300.,-400.,-500. N

Details of "Force"

Scope	
Scoping Method	Geometry Selection
Geometry	5 Faces
Definition	
Type	Force
Define By	Components
Applied By	Surface Effect
Coordinate System	Global Coordinate System
X Component	-300. N (ramped)
Y Component	-400. N (ramped)
Z Component	-500. N (ramped)
Suppressed	No

The Mechanical Scripting Editor (ACT Console)

- Ok. Let’s automate a rather tedious repetitive task
- Imagine that we want to create individual “deformed” STL files of the surfaces shown below (highlighted red and stored in named selection “mysurfaces”)
- There are 20 such surfaces.



Note:
We won’t bother using the record feature, as exporting results do not yet get recorded

- With no way to automate this procedure, we would have to create 20 indivdual surface deformation plots “by hand”
- We would then have to go through each one and “Export” using the “STL file” option
- This would be partcularly cumbersome if we had to run this model several times...

The Mechanical Scripting Editor (ACT Console)

- ...or, you could just cut-and-paste the lines of code below into the editor (or “Open Script”, navigate to the project’s user_files folder, and select “STLExport.py”) and “Run Script”
- Make sure to first clear the editor and shell (if needed) as before

```
expsurfaces = Tree.Find(name="mysurfaces")[0].Location
facelist = expsurfaces.Ids
myresults = []
solu = Model.Analysis[0].Solution
for face in facelist:
    selInfo = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
    selInfo.Ids.Add(face)
    selection = ExtAPI.SelectionManager.NewSelection(selInfo)
    myresults.append(solu.AddTotalDeformation())

solu.EvaluateAllResults()
counter = 1
for result in myresults:
    filname = "C:\\users\\dface"+str(counter)+".stl"
    result.ExportToSTLFile(filname)
    counter += 1
```

The Mechanical Scripting Editor (ACT Console)

- The script creates a deformation plot for each surface, and exports an appropriate STL for each...

load1.py : Description

Loop over surfaces of named selection and export deformed STL file for each...

```
expsurfaces = Tree.Find(name="mysurfaces")[0].Location
facelist = expsurfaces.Ids
myresults = []
solu = Model.Analyses[0].Solution
for face in facelist:
    selInfo = ExtAPI.SelectionManager.CreateSelectionInfo(Sel
    selInfo.Ids.Add(face)
    selection = ExtAPI.SelectionManager.NewSelection(selInfo)
    myresults.append(solu.AddTotalDeformation())
    
solu.EvaluateAllResults()
counter = 1
for result in myresults:
    filname = "C:\\users\\dface"+str(counter)+".stl"
    result.ExportToSTLFile(filname)
    counter += 1
```

- 20 deformation plots created
- 20 STL files created

The Mechanical Scripting Editor Resources

- In STLEXport.py, notice that we're creating the result objects in a separate loop from the export
- While not strictly necessary, we've done this because, once a results object is defined, it has to be evaluated. And result objects can't be evaluated individually (it's all or nothing). Because of this requirement, we thought it's probably more efficient to evalute once –instead of in a loop
- Note the two highlighted lines of code. We encontered this before with the recording tool (slide 9), but this time, we're doing it within a loop: selecting each individual face of the named selection "mysurfaces". This has to be done in order to associate each individual face with it's own deformation result (as required for separate deformed STL files)
- But how did we know where to the find the 'ExportSTLFile' method...?

```
1
2  expsurfaces = Tree.Find(name="mysurfaces")[0].Location
3  facelist = expsurfaces.Ids
4  myresults = []
5  solu = Model.Analyses[0].Solution
6  for face in facelist:
7      selInfo = ExtAPI.SelectionManager.CreateSelectionInfo(SelectionTypeEnum.GeometryEntities)
8      selInfo.Ids = [face]
9      selection = ExtAPI.SelectionManager.NewSelection(selInfo)
10     myresults.append(solu.AddTotalDeformation())
11
12 solu.EvaluateAllResults()
13 counter = 1
14 for result in myresults:
15     filname = "C:\\users\\dface"+str(counter)+".stl"
16     result.ExportToSTLFile(filname)
17     counter += 1
```

The Mechanical Scripting Editor Resources

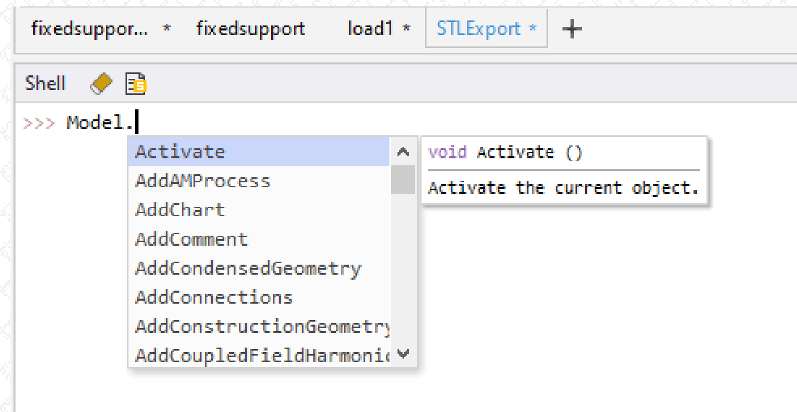
- As you may suspect, success in ACT scripting depends on how successful one is in finding the relevant objects and functionaliy.
- There are three main ways to do this (beyond the record feature we've already seen). We'll list them in the rough order of how frequently we use them (from most to least):

1. Use the object inspection capability of the smart shell. If you don't know what arguments a particular method takes, or even where a method is, this can be useful
2. See the documentation
3. Google it!

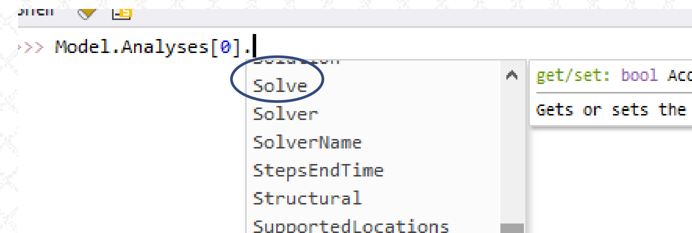
The Mechanical Scripting Editor Resources

1. Object Inspection

- If you type one of the predefined objects from the entry point (see slide 6) into the shell, followd by a dot <.>, all the associated methods and attributes will be listed. For example, if we type ‘Model.’ into the window, we get this...



- When you select any of the of the methods in the drop-down list, you will see a short description of the method, as well as the arguments, if any, it takes
- For example. How do we solve a model in ACT?
- We might suspect that this would be a method under Model.Analyses. We can look for it like this...

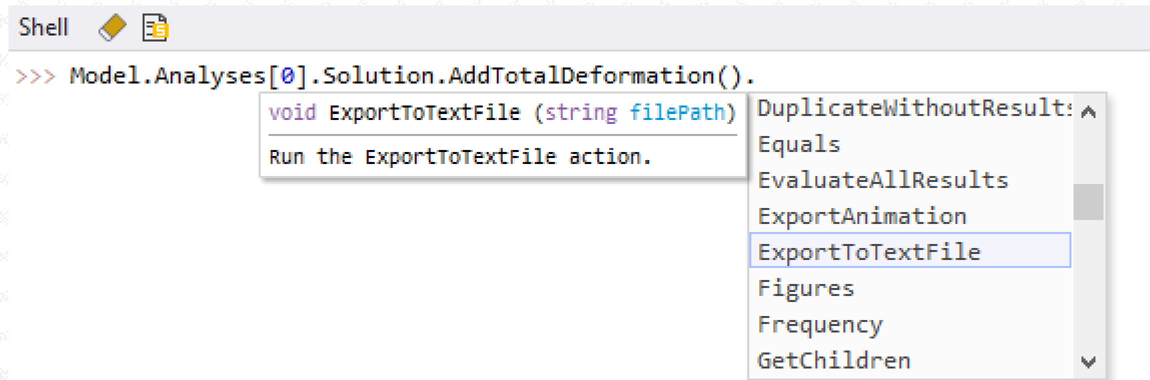


- Scroll down the alphabetical list, or type”Solve”

The Mechanical Scripting Editor Resources

2. See the Documentation

- You will find the documentation invaluable. It grows with each release
- It's how we found the STL export method. We first used shell object insepction to search the methods under result object...



- It wasn't there (we think there's another way to do it under 'Graphics', however)

- so, then we went to the documentation, and found this (here's a link: Export a Result Object to an STL File (ansys.com))

Export a Result Object to an STL File

Goal: Export a result object to an STL file.

Code:

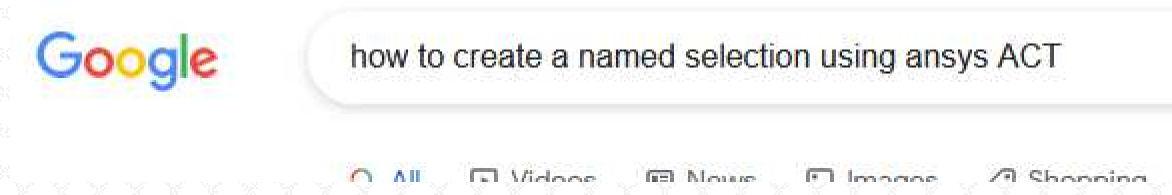
```
result = Model.Analyses[0].Solution.Children[1]
result.ExportToSTLFile("E:\\test.stl")
```

« DUPLICATE AN HARMONIC RESULT OBJECT   EXPORT RESULT IMAGES TO FILES »

The Mechanical Scripting Editor Resources

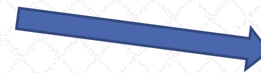
2. Google It!

- Googling turns up a surprising amount of helpful, free material, including tutorials, blog posts, and other documentation to help answer specific questions.
- For example, the following query...



...yielded this...

https://forum.ansys.com/discussion/9548/act-script-assign-items-to-a-named-selection



```
#Model
model = ExtAPI.DataModel.Project.Model

# Get parts
Parts = model.Geometry.GetChildren(DataModelObjectCategory.Part,True)

# The procedure will be included in a loop later. Just using first part in test.
MyPart = Parts[0]

# Selection
SlMn = ExtAPI.SelectionManager
SlMn.ClearSelection()

# ------->>> Missing part: Adding "MyPart" to current selection <<<------
# SlMn.NewSelection()

# Create named selection
NSn = model.NamedSelections
MyNS = NSn.AddNamedSelection()
MyNS.Name = "Part0"
MyNS.Location = SlMn.CurrentSelection
```

Conclusions

- The Mechanical Scripting Editor provides users with a very convenient way to automate cumbersome or repetitive tasks in Ansys Mechanical
- The recent addition of the ‘record’ feature makes this easier than ever before
- As useful as the ‘record’ feature is, users should always rename variables created in this way to suit their application. They should also remove references to DataModel ID’s whenever possible, as these may not be persistent as the model changes
- Users can also find a wealth of documentation and tutorials –starting with the documentation that ships with Ansys products (and may be found on the Customer Portal).