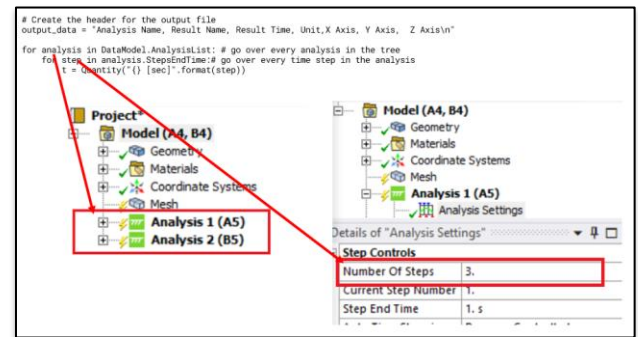


Twin Cities ANSYS® User Meeting

October 26th 2022

Workbench Scripting



... within Epsilon

- Epsilon FEA provides engineering analysis
 - Began in 2008 in Minneapolis
- Making Simulation Accurate
 - In-depth knowledge of the tools
 - ANSYS® Suite of Multi-Physics software
 - Experience with industry successes/failures
 - Aerospace, Rotating Machinery, Electronics, Manufacturing, Packaging, etc.
- Making Simulation Affordable
 - We use specialized experienced engineers
 - Detailed statements of work, scope and budget tracking
 - Automation (APDL, ACT, Journaling, Scripting)
 - NTE / Fixed-Price and with Low hourly rates



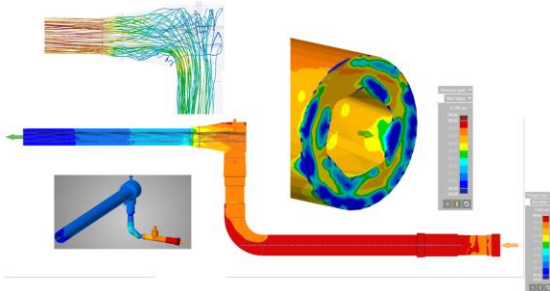
- Our customers need added expertise or load-leveling with:

- Analyst is a team-member, open communication not a black-box
 - Interface with same Epsilon analyst to leverage past experiences
- Any new FEA methods / lessons learned are well communicated
- Schedule/budget fidelity with frequent status updates

Critical!

- Our customers benefit from external expertise

- We infuse up-to-date FEA methods/tools
- We share our knowledge, files, and lessons learned!
- We help with tool selection, infrastructure advice



CFD, Design, Fatigue / Crack propagation, Dynamics / Vibration / Shock, Acoustics, Thermal, Optimization, Scripting / Automation
LS-Dyna / Explicit, and more...

- 8 years of ANSYS Mechanical experience
 - Primarily fatigue assessment on large weldments
 - Wrote my first ANSYS script using the JScript interface to generate a Mechanical model from Excel
 - Started using ANSYS python when ACT console was introduced
- 6 years of general Python experience
 - Mostly in data analysis, image processing, and machine learning

- 1) What is Python?
- 2) Getting into the interface
- 3) Simple script
 - a) Create a load and result.
 - b) Export reaction force and screenshot
- 4) Advanced script
 - a) Find all nodes that have stress above a given level and add their elements to a named selection
- 5) Learning Resources
 - a) python.org
 - b) ANSYS example scripts
 - c) Copies of these and other scripts can be found at <https://github.com/denck007/ANSYS-Scripts>

- Interpreted, object oriented, dynamically typed language
 - Does not have a compile step, just hit run
 - Do not have to declare types for variables, they are dynamic
 - Everything is an object
 - Significant whitespace, no ';' or '{}' are needed like in most other languages. A loop or if ends at the end of it's indented block.
- Python has many implementations
 - Most common is called 'cpython' and is what you get at python.org or from Anaconda
 - ANSYS uses version implemented on the .NET framework called IronPython

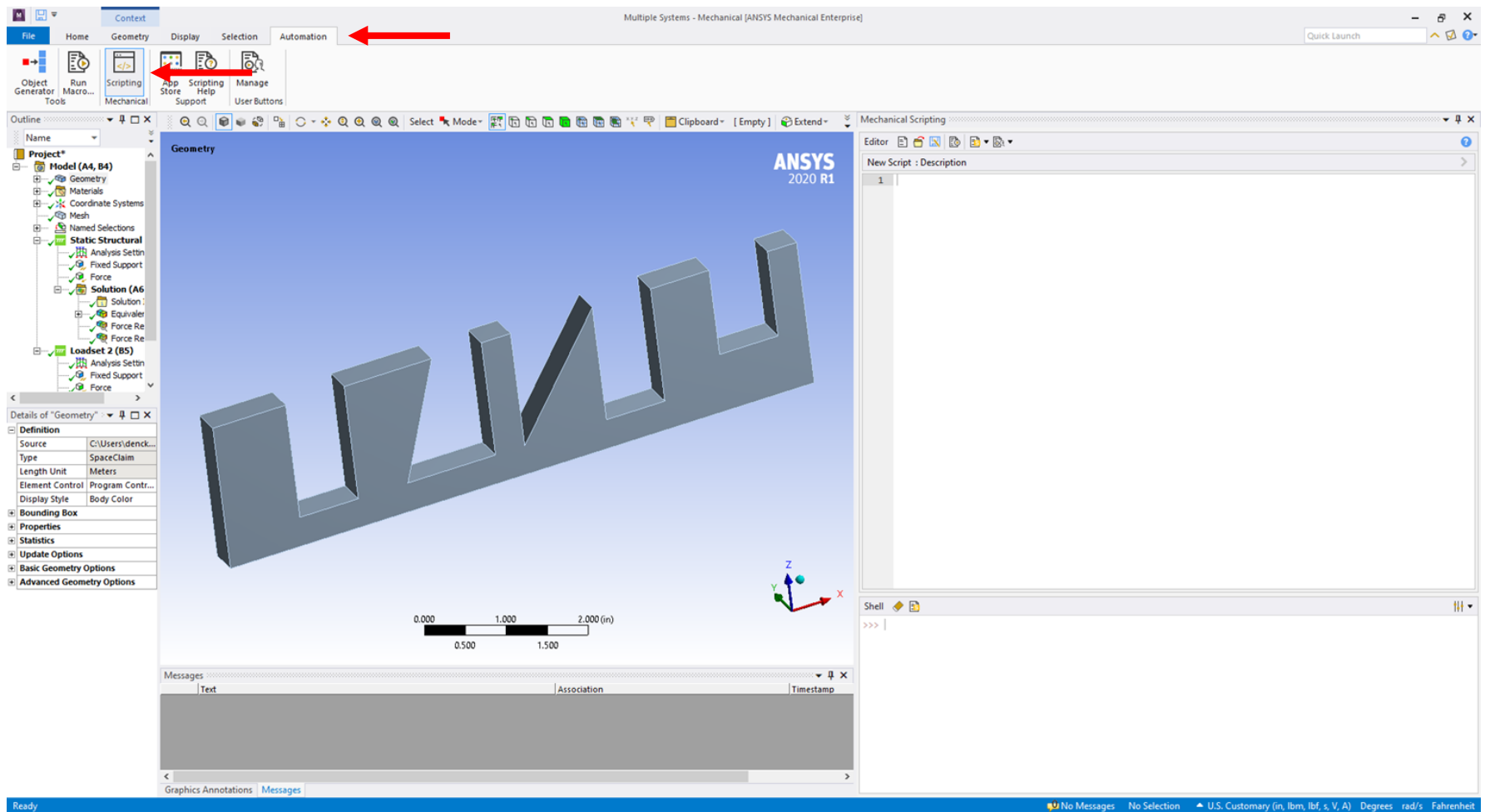


APDL

- Great at interacting with the underlying data
- Called from 'Command' object in the tree to manipulate the underlying FE model
- Very fast math libraries

Python

- Easy to automate GUI workflow in Mechanical
- Easy to interact with CLR programs like Word, Excel, Powerpoint
- Uses the .NET Math libraries (fast, but not Fortran fast)
- Can be invoked and used anywhere and anytime

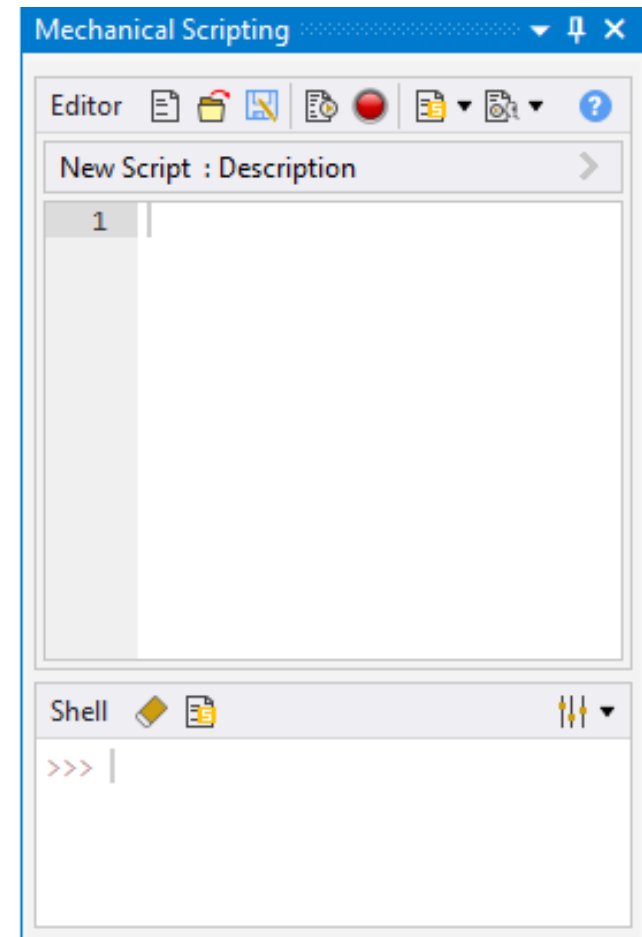


Editor Window (left to right)

- Scripting window
- New blank script (clears the script window)
- Save script to disk
- Load script from disk
- Replay recording
- Start/Stop recording
 - Available if 'Journaling' is turned on
- Save/Load snippet
 - Snippets are scripts stored in App Data
- Save script to be button in UI

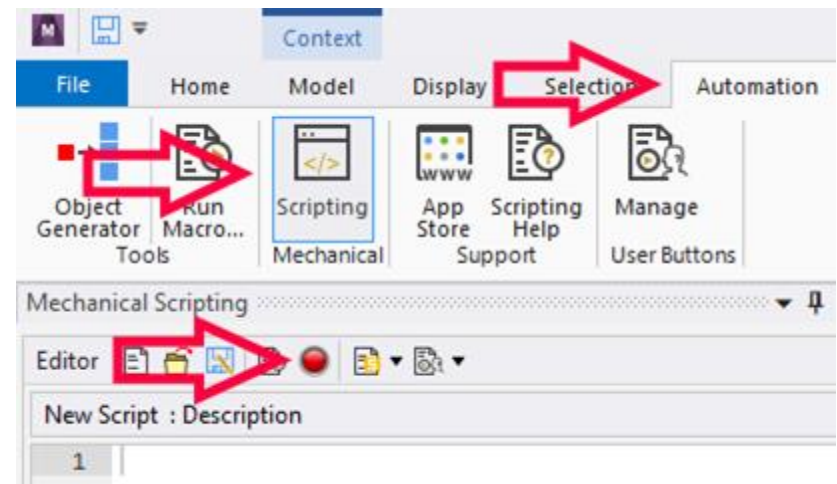
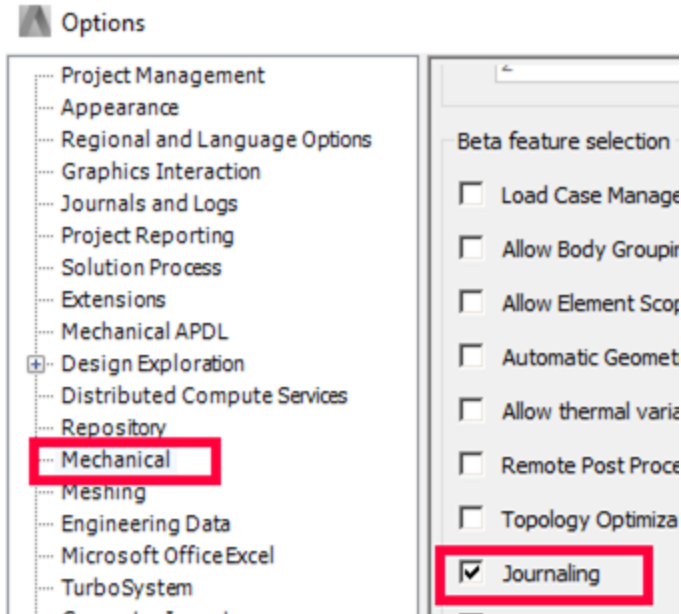
Shell Window (left to right)

- Shell/Terminal
- Clear shell
- Insert various helpful snippets



Journaling - Optional

- Workbench -> Tools -> Options -> Mechanical -> Journaling
- Beta features must be on and the options menu reopened to set this
- Workbench must be restarted after change.
- Every command executed in the UI has the corresponding python command saved.
- Will significantly slow down Mechanical when turned on



- Motivation: Monitoring reaction forces is a simple way to check that the model is behaving as expected.
- Current Workflow: Select each force probe, select all time steps in result, copy and paste into spreadsheet
 - Pros:
 - 1 object in tree for each boundary condition (all timesteps are in tabular data)
 - Cons:
 - Manual copy-paste into spreadsheet, repetitive, easy to mix up order
 - Tedious, takes 5-10 seconds per result
- Proposed Workflow: Script that exports all probes from all analyses into CSV
 - Pros:
 - Executes the exact same every time
 - Takes a fraction of a second to run
 - Cons:
 - Need a probe for each time step for each boundary condition

Example - Export Reaction Forces

```
# Create the header for the output file
output_data = "Analysis Name, Result Name, Result Time, Unit,X Axis, Y Axis, Z Axis\n"

for analysis in DataModel.AnalysisList: # go over every analysis in the tree
    for step in analysis.StepsEndTime: # go over every time step in the analysis
        t = Quantity("{} [sec]".format(step))
        for result in analysis.Solution.Children: # Search the results for reactions matching the time step
            # Check to see if the result is a reaction force probe
            # The 'not' keyword is a boolean inversion
            # The 'continue' keyword skips any following code and goes to the next iteration of the loop
            # Done this way to avoid a bunch of nested if statements
            if not result.GetType().Equals(Ansys.ACT.Automation.Mechanical.Results.ProbeResults.ForceReaction):
                continue

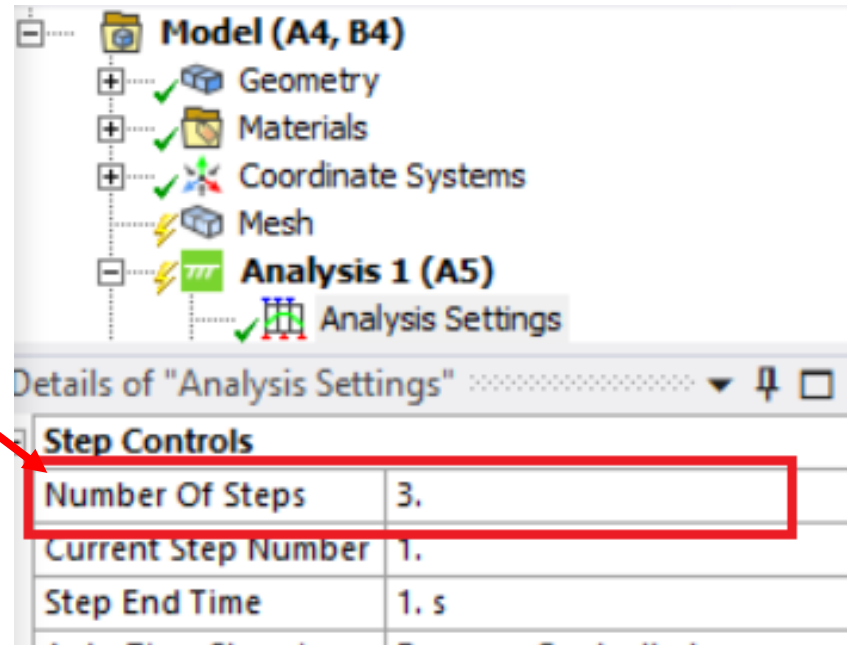
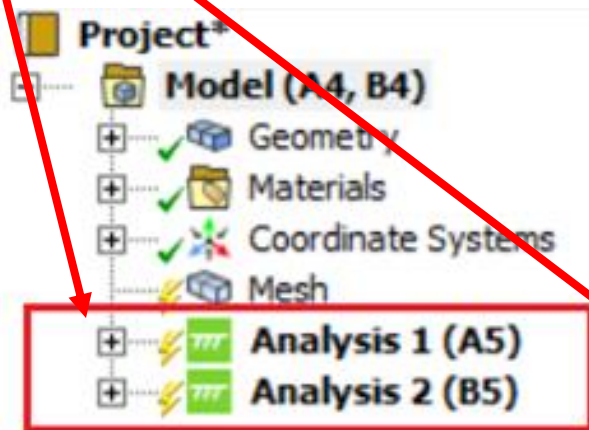
            # Need to handle case where DisplayTime is last. To get the 'Last' a value of '0' is set as the
            # display time. Check for this and set the correct time.
            result_time = result.DisplayTime
            if result.DisplayTime.Equals(Quantity("0 [sec]")):
                # Get the last time and set the result_time
                last_time = analysis.StepsEndTime[analysis.StepsEndTime.Count-1]
                result_time = Quantity("{} [sec]".format(last_time))
            if result_time.Equals(t):
                msg = Ansys.Mechanical.Application.Message("Exporting reaction {} for time {} in analysis
{}".format(result.Name,result_time,analysis.Name),MessageSeverityType.Info)
                ExtAPI.Application.Messages.Add(msg)
                #Results are type Quantity which will print out with the unit which we do not want
                x,unit = result.XAxis.ToString().split(" ") # split at a space, returns number and unit
                y = result.YAxis.ToString().split(" ")[0] # split at a space, keep just the first part which is the number
                z = result.ZAxis.ToString().split(" ")[0] # split at a space, keep just the first part which is the number
                output_data += "{}.{}.{}.{}.{}.{}.{}\n".format(analysis.Name,result.Name,result_time,unit,x,y,z)

# Open the file in write mode ('w' is write, 'r' is read , 'a'is append)
# keywork with is a context manager. It will automatically close the file at the end of the indented block
with open("C:\\temp\\forces.csv",'w') as fp:
    fp.write(output_data)
```

Example - Export Reaction Force

```
# Create the header for the output file
output_data = "Analysis Name, Result Name, Result Time, Unit,X Axis, Y Axis, Z Axis\n"
```

```
for analysis in DataModel.AnalysisList: # go over every analysis in the tree
    for step in analysis.StepsEndTime:# go over every time step in the analysis
        t = Quantity("{} [sec]".format(step))
```



Example - Export Reaction Force

for result in analysis.Solution.Children: # Search the results for reactions matching the time step

Check to see if the result is a reaction force probe

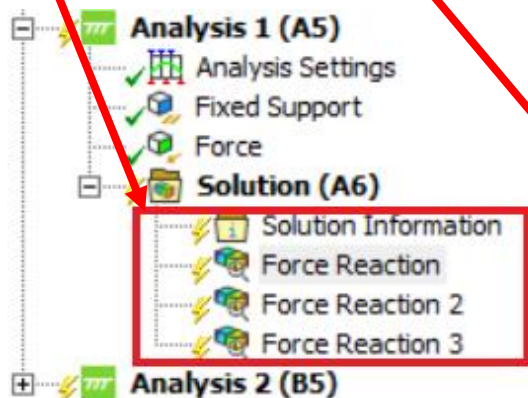
The 'not' keyword is a boolean inversion

The 'continue' keyword skips any following code and goes to the next iteration of the loop

Done this way to avoid a bunch of nested if statements

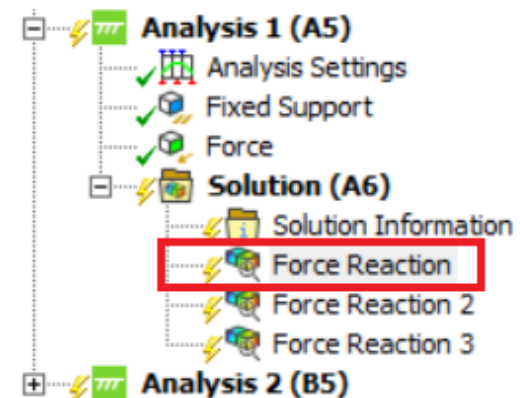
if not result.GetType().Equals(Ansys.ACT.Automation.Mechanical.Results.ProbeResults.ForceReaction):

continue



Details of "Force Reaction" ▾ 🔍 □

| | |
|---------------------------------------|--------------------------|
| Definition | |
| Type | Force Reaction |
| Location Method | Boundary Condition |
| Boundary Condition | Fixed Support |
| Orientation | Global Coordinate System |
| Suppressed | No |
| Options | |
| Result Selection | All |
| <input type="checkbox"/> Display Time | 1. s |



Details of "Force Reaction" ▾ 🔍 □

| | |
|---------------------------------------|--------------------------|
| Definition | |
| Type | Force Reaction |
| Location Method | Boundary Condition |
| Boundary Condition | Fixed Support |
| Orientation | Global Coordinate System |
| Suppressed | No |
| Options | |
| Result Selection | All |
| <input type="checkbox"/> Display Time | 1. s |

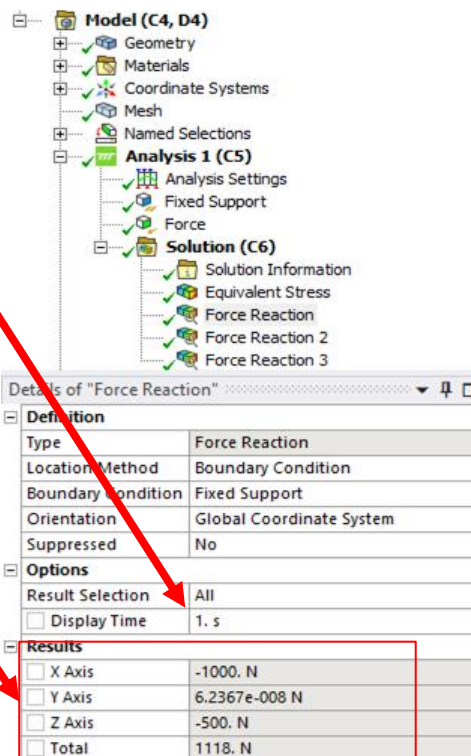
```
# Need to handle case where DisplayTime is last. To get the 'Last' a value of '0' is set as the  
# display time. Check for this and set the correct time.  
result_time = result.DisplayTime  
if result.DisplayTime.Equals(Quantity("0 [sec]")):  
    # Get the last time and set the result_time  
    last_time = analysis.StepsEndTime[analysis.StepsEndTime.Count-1]  
    result_time = Quantity("{} [sec]".format(last_time))
```

- By default, Display Time is set to 'End Time', which is stored as '0'. If Display Time is set to '0 [sec]' know we want the last time step
- Do a check the time to make sure we export in a specific order

Example - Export Reaction Force

```

if result_time.Equals(t):
    msg = Ansys.Mechanical.Application.Message("Exporting reaction {} for time {} in analysis
    {}".format(result.Name,result_time,analysis.Name),MessageSeverityType.Info)
    ExtAPI.Application.Messages.Add(msg)
    #Results are type Quantity which will print out with the unit which we do not want
    x,unit = result.XAxis.ToString().split(" ") # split at a space, returns number and unit
    y = result.YAxis.ToString().split(" ")[0] # split at a space, keep just the first part which is the number
    z = result.ZAxis.ToString().split(" ")[0] # split at a space, keep just the first part which is the number
    output_data += "{}{},{},{},{},{},{}\n".format(analysis.Name,result.Name,result_time,unit,x,y,z)
    
```



Example - Export Reaction Force

```
# Open the file in write mode ('w' is write, 'r' is read, 'a' is append)
# keyword with is a context manager. It will automatically close the file at the end of the indented block
with open("C:\\temp\\forces.csv", 'w') as fp:
    fp.write(output_data)
```

- The with command starts a context manager block.
- Open takes in the file name and the mode to operate on, 'w'rite (overwrites), 'r'ead, 'a'ppend. Additional options exist for binary files.
- Resulting file in Excel (no formatting applied):
 - Note that the Mechanical GUI gives 5 significant figures, this gives access to the unformatted data. This can be useful in cases like finding gaps between points.

| Analysis Name | Result Name | Result Time | Unit | X Axis | Y Axis | Z Axis |
|---------------|------------------|-------------|------|-------------|--------------|--------------|
| Analysis 1 | Force Reaction | 1 [sec] | [N] | -1000 | 6.24E-08 | -500 |
| Analysis 1 | Force Reaction 2 | 2 [sec] | [N] | 400.0000017 | -931.9999993 | 200.0000012 |
| Analysis 1 | Force Reaction 3 | 3 [sec] | [N] | -1.85E-10 | 8.72E-08 | -300 |
| Analysis 2 | Force Reaction | 1 [sec] | [N] | -6.08E-09 | 2.91E-06 | -9999.999998 |

Does the same thing as v1 but does it without needing to have the result object

Example - Export Reaction Forces - Version 2

```
def get_nodes_in_supports(analysis, support_types):
    """
    Given an analysis object and the types of supports to export results for
    Create a dictionary mapping the support name to the node ids in the support
    """
    nodes = {}
    meshObj = analysis.MeshData
    for support_type in support_types:
        for child in analysis.GetChildren(support_type, True):
            nodes[child.Name] = []
            for location_id in child.Location.Ids:
                nodes[child.Name].extend(meshObj.MeshRegionById(location_id).NodeIds)
    return nodes

def read_results_for_analysis(analysis, nodes):
    """
    Given an analysis and a dictionary of {<support name>: <nodes in support>}
    Create a list of results for each support at each time step
    """
    results = []
    for support_name, node_ids in nodes.items():
        with analysis.GetResultsData() as reader:
            times = reader.ListTimeFreq # [1, 2, 3]
            for idx in range(reader.ResultSetCount):
                reader.CurrentResultSet = idx+1 # ResultSet is indexed starting at 1, not 0
                force = reader.GetResult("F")
                result = {
                    "analysis_name": analysis.Name,
                    "support_name": support_name,
                    "time": times[idx],
                    "unit": {},
                    "quantity_type": {},
                    "data": {},
                }

                for component in force.Components:
                    result["unit"][component] = force.GetComponentInfo(component).Unit
                    result["quantity_type"][component] =
                    force.GetComponentInfo(component).QuantityName
                    force.SelectComponents([component])
                    result["data"][component] = sum(force.GetNodeValues(node_ids))
                results.append(result)
    return results
```

```
def get_unique_keys(data, key):
    """
    Given a list of dictionaries data with format:
    [
        {<key>: {"X": 0, "Y": 323, "Z": 43, "ASDF": -1}},
        {<key>: {"X": 0, "Y": 323, "Z": 43, "QWE"}},
    ]
    return: list of unique keys in 'key': ["ASDF", "QWE", "X", "Y", "Z"]
    """
    keys = set()
    for row in data:
        keys.update(row[key].keys())
    return sorted(list(keys))

def get_output_string(results):
    """
    Convert the list of results to a csv style string
    which each component fully listed
    """
    components = get_unique_keys(results, "data")
    output = "Analysis Name,Support Name,Time," + ",".join(["Result {}".format(c) for c in
    components]) + "," + ",".join(["Unit {}".format(c) for c in components]) + "," + ",".join(["Quantity
    {}".format(c) for c in components]) + "\n"
    for result in results:
        line = [
            result["analysis_name"],
            result["support_name"],
            "{}".format(result["time"])
        ]
        line.extend(["{}".format(result["data"][c]) for c in components])
        line.extend(["{}".format(result["unit"][c]) for c in components])
        line.extend(["{}".format(result["quantity_type"][c]) for c in components])

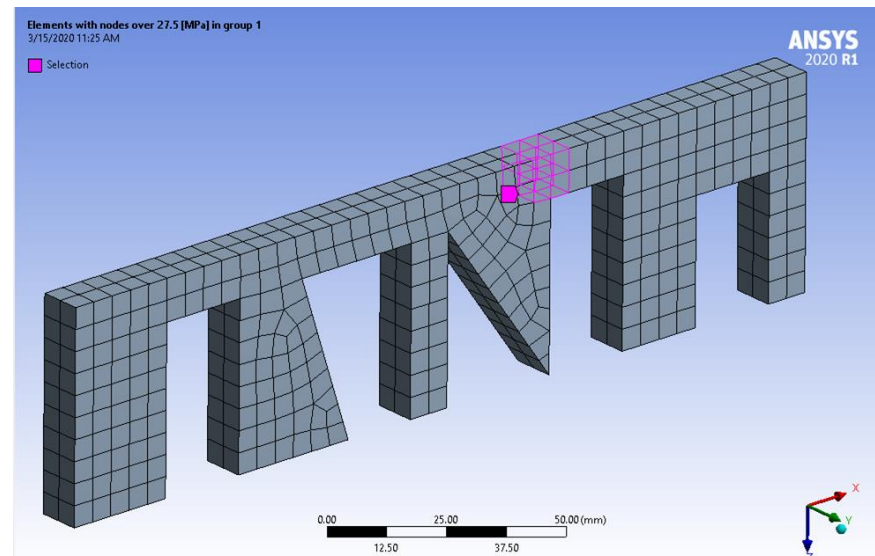
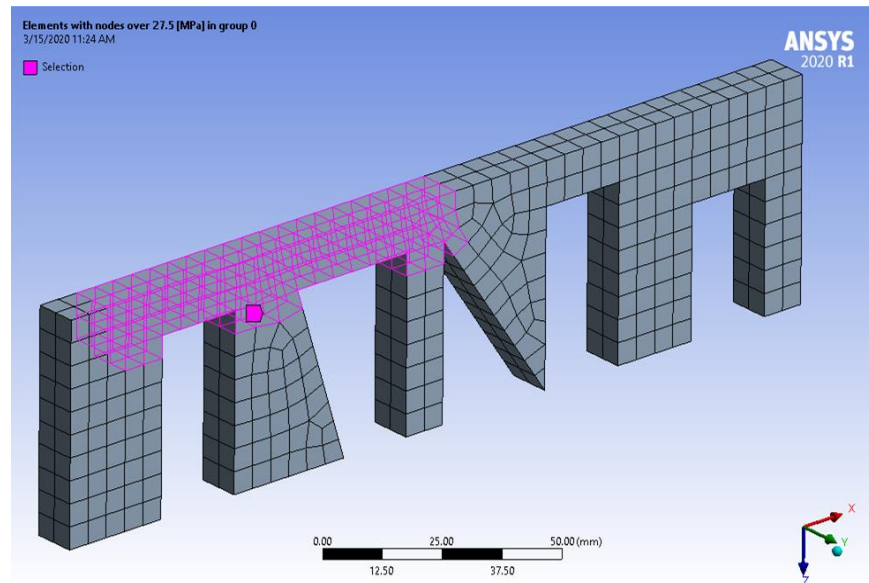
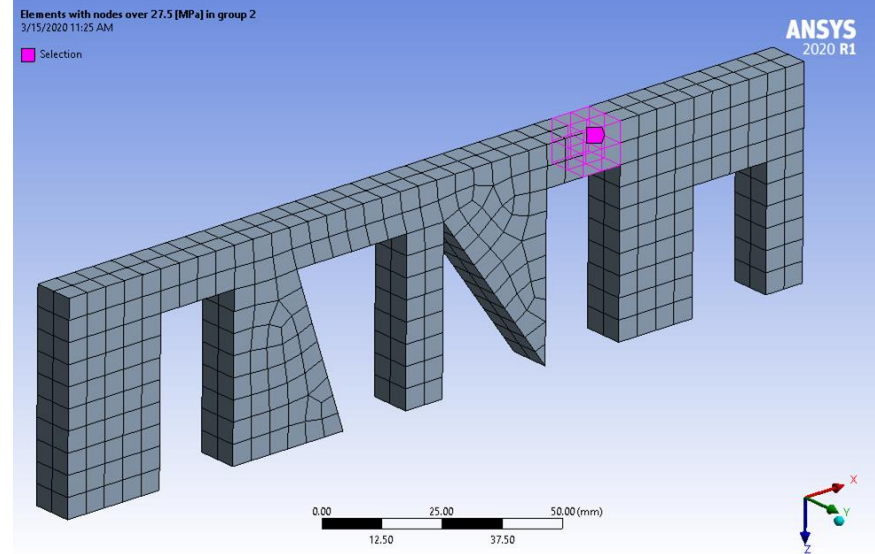
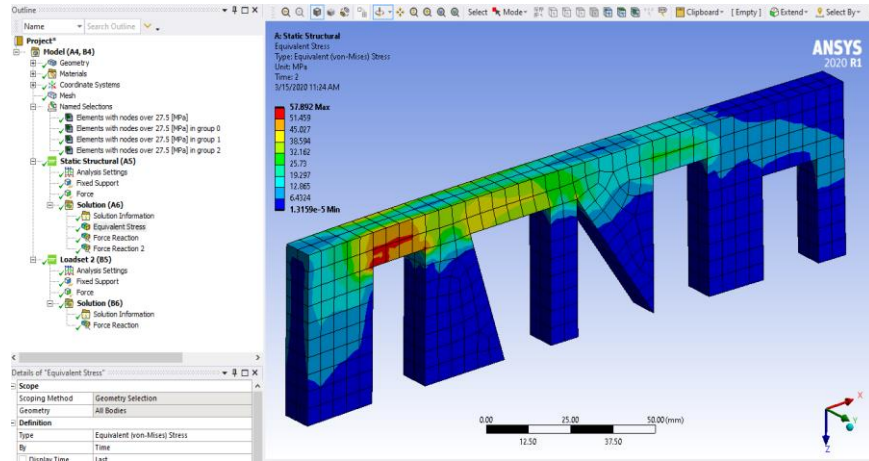
        output += ",".join(line)
        output += "\n"
    return output

support_types = [DataModelObjectCategory.FixedSupport,
DataModelObjectCategory.Displacement, DataModelObjectCategory.RemoteDisplacement]
results = []
for analysis in ExtAPI.DataModel.AnalysisList:
    nodes = get_nodes_in_supports(analysis, support_types)
    results.extend(read_results_for_analysis(analysis, nodes))
    output = get_output_string(results)

    with open("C:\\temp\\forces.csv", 'w') as fp:
        fp.write(output)
```

- Motivation: Review all regions with high stress elements
- Current Workflow: Can use new 'Create Local Probes' feature. Otherwise must do via manual inspection
 - Pros:
 - Manual inspection leads to familiarity with model
 - Cons:
 - Tedious
 - Time consuming
 - Hard to track over time, compounded if there are multiple regions
- Proposed Workflow: Script that creates named selection of all nodes over a specified stress, then groups them into connected regions
 - Pros:
 - Creates a list of regions to inspect
 - All nodes with a high stress are identified.
 - Cons:
 - None

Desired Output High Stress Elements



- Search over all nodes in the model for stresses over the given limit
- Cluster nodes by checking if any nodes on an adjacent element are over the stress limit
 - This is done by finding 'connected components' using 'breadth first search'
- Create a named selection for each of the clusters

```
# Set the stress limit, All elements with nodal stresses over this value will be added to the named selection
stressLimit = Quantity("4000. [psi]") # can use units of MPa, Pa, psi

# Get some of the basic objects needed to perform the work.
sm = ExtAPI.SelectionManager
analysis = DataModel.AnalysisList[0] # This is the first analysis, change the index to run on others
meshObj = analysis.MeshData

# First find the first vonMises stress in the analysis
# If it does not exist throw an error
seqv = None
for child in analysis.Solution.Children:
    if child.GetType().Equals(Ansys.ACT.Automation.Mechanical.Results.StressResults.EquivalentStress):
        seqv = child
if seqv is None:
    msg = Ansys.Mechanical.Application.Message("Did not find a Equivalent Stress result in for analysis {}".format(analysis.Name),MessageSeverityType.Error)
    ExtAPI.Application.Messages.Add(msg)
    # For some reason raise does not work with buttons, but does in console.
    # But it still fails, just in a very unclear way.....
    raise Exception("Did not find a Equivalent Stress result in for analysis {}".format(analysis.Name))

# Next get a listing of the node ids and their corresponding stress value
# Note that the PlotData property does not exist before ANSYS 2020R1
nodeIdsInResult = seqv.PlotData.Values[1]
stressAtNodes = seqv.PlotData.Values[2]
stressUnit = seqv.PlotData.Dependents.Values[0].Unit

# Go over every node and see if it exceeds the stressLimit, and add them to a list
nodesHighStress = []
for node, stress in zip(nodeIdsInResult, stressAtNodes):
    if Quantity("{} {}".format(stress, stressUnit)) > stressLimit:
        nodesHighStress.append(node)

# Convert the node ids from above to their corresponding element ids
elementsHighStress = meshObj.ElementIdsFromNodeIds(nodesHighStress)

# The SelectionManager allows you to do the equivalent of manually selecting
# items in the graphics. It can be scoped to any geometry feature type, or mesh feature type
# First create a SelectionInfo object with all the data that you want to select, then actually
# select the data. This allows for doing many modification to the selection without needing
# to update the graphics which will slow things down
sm.ClearSelection()
selectionInfo = sm.CreateSelectionInfo(SelectionTypeEnum.MeshElements)
selectionInfo.Ids = elementsHighStress
sm.NewSelection(selectionInfo)

# Create the named selection. Just like when using Mechanical normally, anything that is selected
# when a named selection is created is added to the named selection
ns = DataModel.Project.Model.AddNamedSelection()
ns.Name = "Elements with nodes over {}".format(stressLimit.ToString())
```

Epsilon High Stress Elements Continued

```
# This section using Breadth First Search to find connected components
# The underlying theory is beyond the scope of this example.
from collections import deque

def getNeighborElements(element):
    """
    Return the element ids for elements that are adjacent to the element
    """
    nodes = meshObj.NodeIdsFromElementIds([element])
    elements = meshObj.ElementIdsFromNodeIds(nodes)
    # list(set()) gets a list of unique values from a list
    elements = list(set(elements))
    return elements

# Dictionary to allow fast lookup of elements that have high stress
# This also allows mapping of element id to grouping id
# Initialize the tracker with -1 for each element, meaning the element has not been explored
# this is an implementation detail and there are many different ways to approach this
elementTracker = {}
for element in elementsHighStress:
    elementTracker[element] = -1

groups = [] # The element ids
# Now use breadth first search to find groups of high stress elements
for element in elementsHighStress:
    if (element in elementTracker) and (elementTracker[element] == -1):
        # Assign the element to the latest group.
        # The new group is not added to the groups list till all items in the group
        # are found, so the current group is the length of groups, with starting index == 0
        elementTracker[element] = len(groups)
        currentGroup = [element]

        q = deque()
        q.append(element)
        while len(q) > 0:
            elem = q.pop()
            # Add all neighboring elements that have not been explored
            for e in getNeighborElements(elem):
                if (e in elementTracker) and (elementTracker[e] == -1):
                    q.append(e)
                    elementTracker[e] = len(groups)
                    currentGroup.append(e)
        # all the elements in the group have been added to currentGroup, so add
        # the current group to the list of groups
        groups.append(currentGroup)

# Creating named selections behaves the same way as in the UI. Whatever is selected when the NS is created is in the NS
# Of course the values can be edited later if needed
for idx, group in enumerate(groups):
    sm.ClearSelection()
    selectionInfo = sm.CreateSelectionInfo(SelectionTypeEnum.MeshElements)
    selectionInfo.Ids = group
    sm.NewSelection(selectionInfo)
    ns = DataModel.Project.Model.AddNamedSelection()
    ns.Name = "Elements with nodes over {} in group {}".format(stressLimit.ToString(), idx)
```



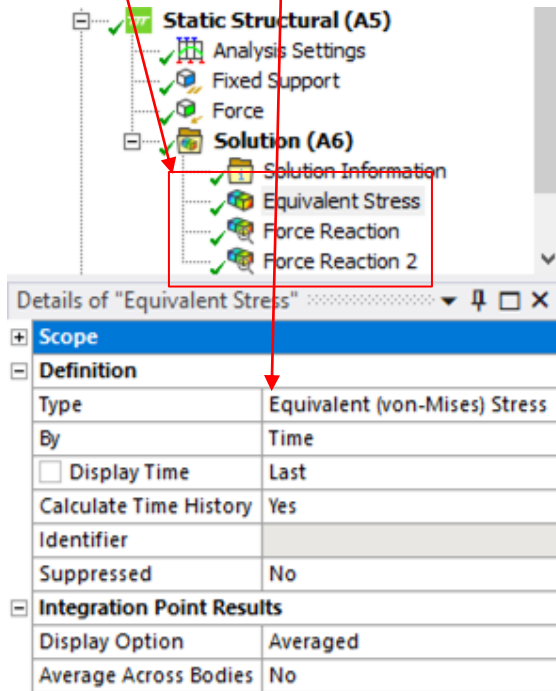
```
# Set the stress limit, All elements with nodal stresses over this value will be added to the named selection
stressLimit = Quantity("4000. [psi]") # can use units of MPa, Pa, psi
```

```
# Get some of the basic objects needed to perform the work.
sm = ExtAPI.SelectionManager
analysis = DataModel.AnalysisList[0] # This is the first analysis, change the index to run on others
meshObj = analysis.MeshData
```

- Create a Quantity object with our desired threshold
 - Useful as it handles all of the units for us (ie can input MPa and have psi be the current units)
- Set shorthand for some of the items we will be using
 - sm is the interface to select objects (like bodies, edges, nodes, elements, etc.)
 - Set the analysis we are looking at
 - Get the object that holds all the mesh data

Epsilon Example - High Stress Elements

```
# First find the first vonMises stress in the analysis
# If it does not exist throw an error
seqv = None
for child in analysis.Solution.Children:
    if child.GetType().Equals(Ansys.ACT.Automation.Mechanical.Results.StressResults.EquivalentStress):
        seqv = child
if seqv is None:
    msg = Ansys.Mechanical.Application.Message("Did not find a Equivalent Stress result in for analysis {}".format(analysis.Name), MessageSeverityType.Error)
    ExtAPI.Application.Messages.Add(msg)
    # For some reason raise does not work with buttons, but does in console.
    # But it still fails, just in a very unclear way.....
    raise Exception("Did not find a Equivalent Stress result in for analysis {}".format(analysis.Name))
```



- Search through the solution and find the vonMises Stress result
- Throw an error if it is not found

```
# Next get a listing of the node ids and their corresponding stress value
# Note that the PlotData property does not exist before ANSYS 2020R1
nodeIdsInResult = seqv.PlotData.Values[1]
stressAtNodes = seqv.PlotData.Values[2]
stressUnit = seqv.PlotData.Dependents.Values[0].Unit

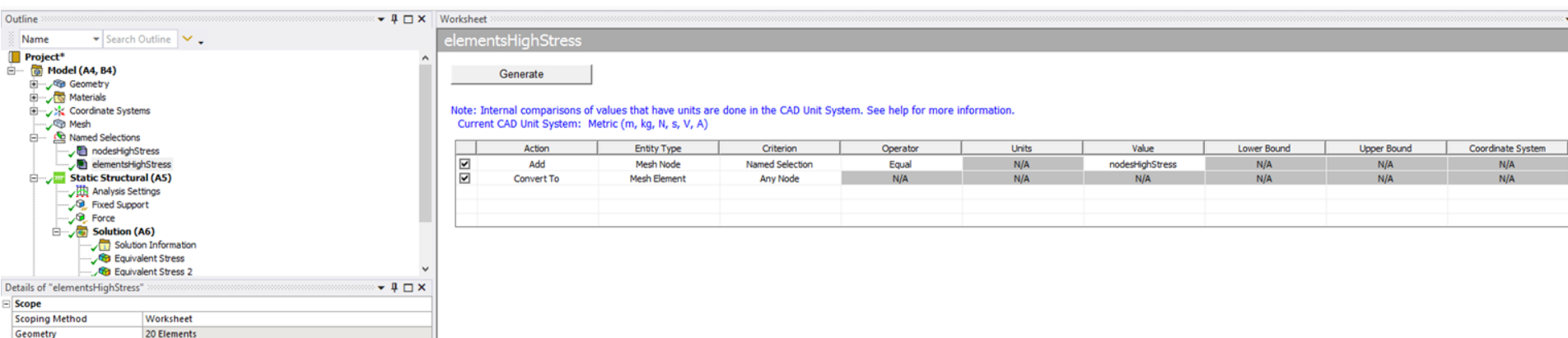
# Go over every node and see if it exceeds the stressLimit, and add them to a list
nodesHighStress = []
for node, stress in zip(nodeIdsInResult, stressAtNodes):
    if Quantity("{} {}".format(stress, stressUnit)) > stressLimit:
        nodesHighStress.append(node)
```

- Iterate over every node in the solution and get a list of all the nodes and stresses at the nodes
 - IMPORTANT: The data is EXACTLY what was requested in the solution. If the plot has 'Display Option' set to 'Averaged' (default) it is 1 value per node. If it is set to 'Unaveraged' there is a result per element node. In this model there are 3561 nodes so 3561 items in the 'Averaged' result, but 10380 values in the 'Unaveraged' result.
- Note that a Quantity object is created for every node and used in the comparison. It would be faster (but less clear) to convert the threshold stress to the PlotData stress units and do the comparison.

Epsilon Example - High Stress Elements

```
# Convert the node ids from above to their corresponding element ids  
elementsHighStress = meshObj.ElementIdsFromNodeIds(nodesHighStress)
```

- Use the meshObj (analysis.MeshData) to convert a listing of mesh nodes to mesh elements
- Can be thought of as creating a named selection of nodesHighStress, and converting it to Mesh Elements



The screenshot displays the ANSYS Workbench interface. On the left, the 'Outline' pane shows a project structure with 'Model (A4, B4)' containing 'Geometry', 'Materials', 'Coordinate Systems', 'Mesh', and 'Named Selections'. 'Static Structural (A5)' is selected, showing 'Analysis Settings', 'Fixed Support', 'Force', and 'Solution (A6)'. 'Solution (A6)' includes 'Solution Information', 'Equivalent Stress', and 'Equivalent Stress 2'. The 'Details of "elementsHighStress"' panel at the bottom left shows 'Scoping Method' as 'Worksheet' and 'Geometry' as '20 Elements'.

The main 'Worksheet' area is titled 'elementsHighStress' and contains a 'Generate' button. Below the button, a note states: 'Note: Internal comparisons of values that have units are done in the CAD Unit System. See help for more information. Current CAD Unit System: Metric (m, kg, N, s, V, A)'. A table with the following columns is displayed: Action, Entity Type, Criterion, Operator, Units, Value, Lower Bound, Upper Bound, and Coordinate System.

| | Action | Entity Type | Criterion | Operator | Units | Value | Lower Bound | Upper Bound | Coordinate System |
|-------------------------------------|------------|--------------|-----------------|----------|-------|-----------------|-------------|-------------|-------------------|
| <input checked="" type="checkbox"/> | Add | Mesh Node | Named Selection | Equal | N/A | nodesHighStress | N/A | N/A | N/A |
| <input checked="" type="checkbox"/> | Convert To | Mesh Element | Any Node | N/A | N/A | N/A | N/A | N/A | N/A |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

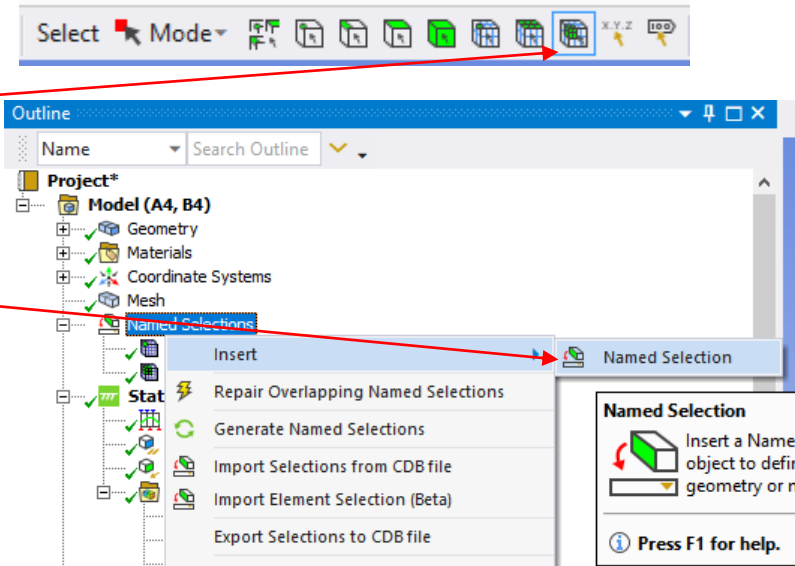
Epsilon Example - High Stress Elements

```
# The SelectionManager allows you to do the equivalent of manually selecting
# items in the graphics. It can be scoped to any geometry feature type, or mesh feature type
# First create a SelectionInfo object with all the data that you want to select, then actually
# select the data. This allows for doing many modification to the selection without needing
# to update the graphics which will slow things down
sm.ClearSelection()
selectionInfo = sm.CreateSelectionInfo(SelectionTypeEnum.MeshElements)
selectionInfo.Ids = elementsHighStress
sm.NewSelection(selectionInfo)
```

```
# Create the named selection. Just like when using Mechanical normally, anything that is selected
# when a named selection is created is added to the named selection
ns = DataModel.Project.Model.AddNamedSelection()
ns.Name = "Elements with nodes over {}".format(stressLimit.ToString())
```

- Create a new selection of elements
- Sets the ids in the selection to be the elements

- Note this same process works with all geometry types
- Multiple 'SelectionInfo' objects can exist at the same time (and can be added/subtracted to), but only 1 'SelectionInfo' object selected at once
- Create the selection (set it to be selected in the GUI) with 'NewSelection'



Epsilon Example - High Stress Elements

```
def getNeighborElements(element):
```

```
'''
```

```
Return the element ids for elements that are adjacent to the element
```

```
'''
```

```
nodes = meshObj.NodeIdsFromElementIds([element])
```

```
elements = meshObj.ElementIdsFromNodeIds(nodes)
```

```
# list(set()) gets a list of unique values from a list
```

```
elements = list(set(elements))
```

```
return elements
```

- Create a function that takes in a single element, and returns a list of all adjacent elements
- Equivalent to the following Named Selection:

ANSYS 2020 R1

Select Nodes on Element

Select Elements From Nodes

Worksheet

Select Elements From Nodes

Generate

Note: Internal comparisons of values that have units are done in the CAD Unit System. See help for more information.
Current CAD Unit System: Metric (m, kg, N, s, V, A)

| | Action | Entity Type | Criterion | Operator | Units | Value | Lower Bound | Upper Bound | Coordinate |
|-------------------------------------|------------|--------------|------------|----------|-------|-------|-------------|-------------|------------|
| <input checked="" type="checkbox"/> | Add | Mesh Elem... | Element ID | Equal | N/A | 2 | N/A | N/A | N/A |
| <input checked="" type="checkbox"/> | Convert To | Mesh Node | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| <input checked="" type="checkbox"/> | Convert To | Mesh Elem... | Any Node | N/A | N/A | N/A | N/A | N/A | N/A |

```
elementTracker = {}  
for element in elementsHighStress:  
    elementTracker[element] = -1
```

- Keep track of which elements to find connected components (aka groups, clusters) by using a dictionary
 - Elements that we are not interested in are not in the dictionary
 - -1 means the element has not been assigned to a group
 - All other values indicate the group number
- Dictionaries are very fast and efficient ways to store key, value pairs.
 - Keys are any immutable data type (tuple, integer, float, string')
 - Values can be any python object (list, number, string, even another dictionary)

```
groups = [] # The element ids
for element in elementsHighStress:
    if (element in elementTracker) and (elementTracker[element] == -1):
        elementTracker[element] = len(groups)
        currentGroup = [element]
        q = deque()
        q.append(element)
        while len(q) > 0:
            elem = q.pop()
            for e in getNeighborElements(elem):
                if (e in elementTracker) and (elementTracker[e] == -1):
                    q.append(e)
                    elementTracker[e] = len(groups)
                    currentGroup.append(e)
        groups.append(currentGroup)
```

- Find 'Connected Components of Graph using Breadth First Search'
 - Note this implementation is not the most efficient and may not scale well to a large number of elements as it checks the same elements multiple times.
- A Queue is a data structure that can efficiently store, return, and insert a value and maintain the order in which items are added. 'pop()' an item returns the most recently added item
- This block of code adds an element to a queue, then adds its neighbors if they are a high stress element. When no more neighboring elements can be found that are high stress, then it moves onto the next grouping of elements.


```
for idx, group in enumerate(groups):  
    sm.ClearSelection()  
    selectionInfo = sm.CreateSelectionInfo(SelectionTypeEnum.MeshElements)  
    selectionInfo.Ids = group  
    sm.NewSelection(selectionInfo)  
    ns = DataModel.Project.Model.AddNamedSelection()  
    ns.Name = "Elements with nodes over {}".format(stressLimit.ToString(),idx)
```

- Finally create a new named selection for each of the groups

- Scripting with python can drastically improve the quality of work by removing some of the tedious and error prone tasks we all do.
- The Python interface is improving every release
- We are here to help!

- Mechanical API documentation

- https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v201/en/act_script/act_mech_api.html

- ANSYS scripting help

- https://ansyshelp.ansys.com/account/secured?returnurl=/Views/Secured/corp/v201/en/act_script/act_script_intro.html

- Extension Examples

- <https://catalog.ansys.com/Developers.cshtml>
- Download 'Extension Examples <your version>' More example scripts than what is in documentation.
- Example 'Mises' is especially helpful for working with stress results

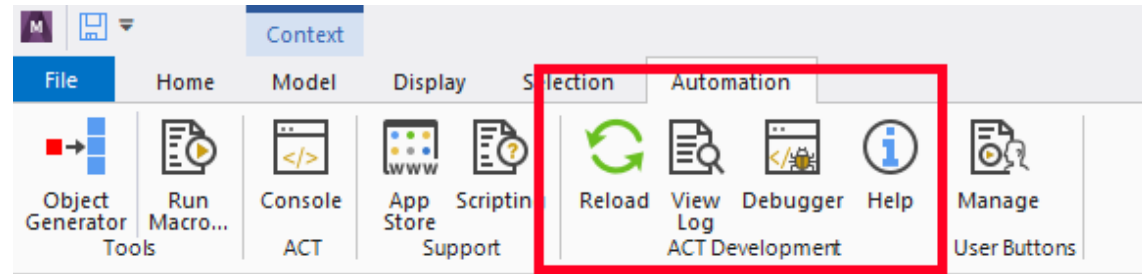
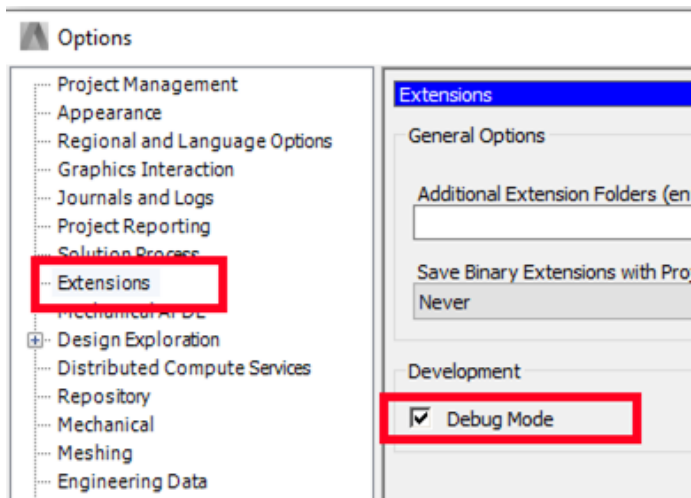
- Python

- Note ANSYS Mechanical currently uses python 2.7 which is deprecated. Official tutorials: <https://docs.python.org/2.7/tutorial/>

- Neil Dencklau's Github

- Has the scripts for this presentation and additional scripts showing other use cases
- <https://github.com/denck007/ANSYS-Scripts>

- Workbench -> Tools -> Options -> Extensions -> Debug
- Loads ACT Debugger, Log file viewer, Reload Extensions
- Not going over building ACT extensions so the debugger and reload tool are not used. Log file viewer can be useful in scripting.



- Two types of data:
 - What is in the tree - Details, name, timestep, stiffness behavior, etc
 - The underlying representation - BREP data for geometry, nodes and elements, results, etc
- Everything that is in the 'details' of an item in the tree is available
- Most of the underlying data directly accessible (more with every release)

cPython

- Built on C language
- Highly extensible by writing extensions in C or C++
- Lots of libraries
 - numpy for math
 - matplotlib for high quality graphs
 - scipy for more advanced math
 - opencv for image processing
 - tensorflow for machine learning
 - django for web sites
 - 220,000+ other packages available for free on pypi.org
- Has a 'Global Interpreter Lock' which makes multi-threading kind of confusing

IronPython

- Built using Common Runtime Library in .NET
- Use the standard windows math dll libraries that ANSYS is already using
- Easy to call existing .NET libraries
- Easy to add a scripting interface to .NET programs
- Most extensions from cPython do not work
- Multi-threading/multi-processing is easier

Notes

- Show previous line in the ANSYS console
 - CTRL + UpArrow
- Formatting strings using the .format() method
 - "{}".format("some string") => "some string"
 - "{}".format(234.44) => "234.44"
 - "{:.1f}".format(234.44) => "234.4"
 - "{:03d}".format(43) => "043"
- Comments start with a #. Anything after the # will not be executed
- Multi-line comments are enclosed with triple single quotes
 - '''multi-line comment'''
- Having print statements in the code for a button will cause the script to crash. Instead use the following and view the results in the debug log.
 - ExtAPI.Application.Messages.Add(Ansys.Mechanical.Application.Message("message info", MessageSeverityType.Info)
 - MessageSeverityType options are: .Info, .Warning, and .Error

Inputs

```
print("Range Function")
for ii in range(3):
    print(ii)
```

```
print("Iterate over objects in list")
for item in ["item1", "item2", "item3"]:
    print(item)
```

```
print("Break keyword")
for ii in range(3):
    if ii == 1:
        print("ii==1, breaking")
        break
    print(ii)
```

```
print("Continue keyword")
for ii in range(3):
    if ii == 1:
        print("ii==1, do not print value")
        continue
```

```
print("While Loop")
ii = 0
while ii < 3:
    print(ii)
    ii += 1
```

Outputs

Range Function

0
1
2

Iterate over objects in list

item1
item2
item3

Break keyword

0
ii==1, breaking

Continue keyword

0
ii==1, do not print value
2

While Loop

0
1
2

Inputs

```
if True:
    print("Item is True")
else:
    print("Item is False")
```

```
if "nsy" in "ansys":
    print("True")
```

```
x = 3
if x > 5:
    print("{} Greater than 5".format(x))
elif x > 3:
    print("{} Greater than 3".format(x))
elif x >= 3:
    print("{} Greater than or equal to 3".format(x))
else:
    print("Did not find match")
```

Outputs

Item is True

True

3 Greater than or equal to 3





... within Epsilon