PYTHON SCRIPTS FOR ABAQUS

LEARN BY EXAMPLE

Gautam Puri

This document is a preview of the book.

Book website: www.abaquspython.com

Dedicated to Mom

First Edition 2011

Disclaimer: The author does not offer any warranties for the quality or validity of the information contained in this book or the included example Python scripts. This book has been written for entertainment purposes only and should be treated as such. The reader is responsible for the accuracy and usefulness of any analyses performed with the Abaqus Software, with or without the use of Python scripts. The reader is also responsible for the accuracy and usefulness of any non-Abaqus related Python programs or software developed. The information contained in the book is not intended to be exhaustive or apply to any particular situation and must therefore be viewed with skepticism and implemented with extreme caution. The Python scripts available with this book have been included for their instructional value. They have been tested with care but are not guaranteed for any particular purpose. In no event shall the author be liable for any incidental, indirect or consequential damages arising from use of this book or the example scripts provided with it.

In plain English, by reading this document you acknowledge that the author is not responsible for your finite element studies, nor is he responsible for the validity of their results or their interpretation.

Printed in the United States of America

Book website: www.abaquspython.com

Contents

This preview contains snippets from each of the 22 chapters. No table of contents is available for the preview version.

The entire book is approximately 745 pages long; its entire table of contents is available as a separate download on the book website www.abaquspython.com

Preface

If you're reading this, you've probably decided to write a Python script to run an FEA analysis in Abaqus. But you're not sure where to begin, you've never written a working script for Abaqus, and you've never worked with the programming language Python either. The good news is you've found the right book to deal with the situation. Through the course of this text you're going to learn the basics of writing scripts for Abaqus and understand the working of the Abaqus Scripting Interface. At the same time you're going to learn what you need to know of the Python programming language itself. You're going to receive the stable foundation you need so that you spend more time focusing on your research and less time debugging code.

The aim of this book is not to teach you every single built-in scripting method offered by Abaqus. There are literally hundreds of these, and chances are you will only use a few of them for your own simulations. We'll focus on these, and put you in a position where you can branch out on your own. For the record all the keywords and methods of the Abaqus Scripting Interface are listed in the Abaqus Scripting Reference Manual. The documentation also consists of a manual called the Abaqus Scripting User's Manual which provides helpful advice on different scripting topics. You could potentially learn to write Abaqus scripts in Python from the documentation itself, as many people (such as me) have had to do in the past. But as a beginner you will likely find yourself overwhelmed by the sheer quantity of information provided there. You will spend a lot of time making avoidable mistakes and discovering for yourself, after hours (or days or months) of trial and error, the correct method to accomplish a given task through a script. This book gives you the guidance you need to start writing complex scripts right off the bat. Once you've read through all the pages you will have the knowledge and the confidence to write your own scripts for finite element simulations in Abaqus, and will then be able to refer to the Abaqus documentation for more information specific to your research task.

Why write scripts?

If you plan to learn scripting in Abaqus chances are you already know why it is useful and intend to use it to accomplish some task for your analyses. But for the sake of

completeness (and for those of you who are reading because your professor/boss forced you to), a few uses shall be mentioned.

Let's assume you regularly use a few materials in all your simulations. Every time you start a new simulation in the GUI mode (Abaqus/CAE) you need to open up the materials editor and enter in material properties such as the Density, Young's Modulus, and Poisson's Ratio and so on for each of these materials. You could instead put all of these materials in a script. Then all you would need to do is go to *File* > *Run Script*... and your material database would be populated with these materials in a couple of seconds. Basically you would be using the script to perform a repetitive task to save time. That is the one use of a script, to perform the same task the same way multiple times with minimal effort. We will in fact look at this example of creating materials with a script in the first chapter.

A more complex use of a script is if you have a certain part on which you plan to apply loads and boundary conditions, and you wish to change the loads, constraints, or the geometry of the part itself and rerun the simulation numerous times to optimize the design. Let's assume for example you apply a load on a horizontal cantilevered beam and you want to know how much the beam bends as you increase its length. One way to do this would be to recreate the beam part 7 or 8 times. If your simulation has complex parameters you might have to apply sections, loads and constraints to it every time. A more sophisticated and efficient way to accomplish the same task is to write a script with the length of the beam assigned to a variable. You could then change the value of this variable and rerun the script in a loop as many times as you need to. The script would redraw the beam to the new length and apply the loads and BCs in the correct regions (accounting for the change in location of loads and BCs with the geometry). While this may sound like too much work for a simple beam simulation, if you have a more complex part with multiple dimensions that are all related to each other then remodeling it several times will prove to be very time consuming and a script will be the wise choice.

An added advantage of a script is that you have your entire simulation setup saved in the form of a small readable text file only a few kilobytes in size. You can then email this text file to your coworker and all he would need to do is run this script in Abaqus. It would redraw the part, apply the materials, loads, boundary conditions, create the steps, and even create and run the job if programmed to do so. This also has the advantage of readability. If a coworker takes over your project, he does not need to navigate through

the model tree to figure out how you created the complex geometry of your part file, or what points and edges you applied each load or boundary condition on. He only needs to open up the script file and it's all clearly spelled out. And you can put comments all over the script to explain why you did what you did. It keeps things compact and easy to follow.

What you need…

This book assumes that you have some previous experience with running simulations in Abaqus in the GUI (Abaqus/CAE). This means you know how to set up a basic simulation, create parts, enter material properties, assign sections, apply forces and boundary conditions, create interactions, mesh parts and run jobs by using the toolbars or menus in Abaqus/CAE. When we start learning to write scripts you will essentially be performing all of these same procedures, except in the form of Python code.

However you do not need to be an expert at these tasks. For every example we work on, we first look at the procedure to be carried out in the Abaqus/CAE. This procedure has been spelled out in the text, and is also demonstrated as silent video screencasts where you can watch me perform the analysis step by step. This is to ensure that you know how to perform the task in the GUI itself, before trying to write a script. These screencasts have been posted on the book website www.abaquspython.com (and hosted on YouTube) where I've found they are also being used by beginners trying to teach themselves Abaqus. Following the creation of these videos, I was employed by Dassault Systèmes Simulia Corp. to create an Abaqus tutorial series on their new 'SIMULIA Learning Community'. I have recorded audio narration with detailed explanation over all of these, and other newer tutorials as well. These are currently displayed (free) at www.simulia.com/learning. If you wish to brush up on your Abaqus skills you may watch these. Refer to the book website for up-to-date information and links.

The book assumes that you have some basic knowledge of programming. This includes understanding concepts like variables, loops (for, while) and if-then statements. You are all set if you have experience with languages such as C, C++, Java, VB, BASIC etc. Or you might have picked up these concepts from programmed engineering software such as MATLAB or Mathematica.

In order to run the example scripts on your own computer you will need to have Abaqus installed on it. Abaqus is the flagship product of SIMULIA, a brand of Dassault

Systèmes. If you have Abaqus (research or commercial editions) installed on the computers at your workplace you can probably learn and practice on those. However not everyone has access to such facilities, and even if you do you might prefer to have Abaqus on your personal computer so you can fiddle around with it at home. The good news is that the folks at SIMULIA have generously agreed to provide readers of this book with Abaqus Student Edition version 6.10 (or latest available) for free. It can be downloaded off the book website. This version of Abaqus can be installed on your personal computer and used for as long as you need to learn the software. There are a few minor restrictions on the student edition, such as a limitation on the number of nodes (which means we will not be able to create fine meshes), but for the most part these will not hinder the learning experience. For our purposes Abaqus SE is identical to the research and commercial editions. The only difference that will affect us is the lack of replay files but I'll explain what those are and how to use them so you won't have any trouble using them on a commercial version. Abaqus SE version 6.9 and version 6.10 were used to develop and test all the examples in this book. The Abaqus Scripting Interface in future versions of Abaqus should not change significantly so feel free to use the latest version available to you when you read this.

How this book is arranged…

The first one-third of this book is introductory in nature and is meant to whet your appetite, build up a foundation, and send you in the right direction. You will learn the basics of Python, and get a feel for scripting. You'll also learn essential stuff like how to run a script from the command line and what a replay file is.

The second part of the book helps you 'Learn by Example'. It walks you through a few scripting examples which accomplish the same task as the silent screencasts on the book website but using only Python scripts. Effort has been taken to ensure each example/script touches on different aspects of using Abaqus. All of these scripts create a model from start to finish, including geometry creation, material and section assignments, assembling, assigning loads, boundary conditions and constraints, meshing, running a job, and post processing. These scripts can later be used by you as a reference when writing your own scripts, and the code is easily reusable for your own projects. Aside from demonstrating how to set up a model through a script, the later chapters also demonstrate how to run optimization and parametric studies placing your scripts inside

loops and varying parameters. You also get an in-depth look into extracting information from output databases, and job monitoring.

The last part of the book deals with GUI Customization – modifying the Abaqus/CAE interface for process automation and creating vertical applications. It is assumed that you have no previous knowledge of GUI programming in general, and none at all with the Abaqus GUI Toolkit. GUI Customization is a topic usually of interest only to large companies looking to create vertical applications that perform repetitive tasks while prompting the user for input and at the same time hiding unnecessary and complex features of the Abaqus interface. Chances are most readers will not be interested in GUI Customization but it has been included for the sake of completeness and because there is no other learning resource available on this topic.

Acknowledgements

I would like to thank my mother for giving me the opportunity to pursue my studies at a great expense to herself. This book is dedicated to her. I would also like to thank my father and my grandmother for their love, support and encouragement.

I'd like to thank my high school Physics teacher, Santosh Nimkar, for turning a subject I hated into one I love. The ability to understand and predict real world phenomena using mathematics eventually led me toward engineering.

I'd like to extend a special thank you to Rene Sprunger, business development manager at SIMULIA (Dassault Systèmes Simulia Corporation) for his support and encouragement, without which this book might never have materialized. I'd also like to thank all the professionals at SIMULIA for developing the powerful realistic simulation software Abaqus, and for creating the remarkable Abaqus Scripting Interface to enhance it.

PART 1 – GETTING STARTED

The chapters in Part 1 are introductory in nature. They help you understand how Python scripting fits into the Abaqus workflow, and explain to you the benefits and limitations of a script. You will learn the syntax of the Python programming language, which is a prerequisite for writing Abaqus scripts. You will also learn how to run a script, both from within Abaqus/CAE and from the command line. We'll introduce you to replay files and macros, and help you decide on a code editor.

It is strongly recommended that you read all of these chapters, and do so in the order presented. This will enhance your understanding of the scripting process, and ensure you are on the right track before moving on to the examples of Part 2.

1

A Taste of Scripting

1.1 Introduction

The aim of this chapter is to give you a feel for scripting in Abaqus. It will show you the bigger picture and introduce you to idea of how a script can replace actions you would otherwise perform in graphical user interface (GUI) Abaqus/CAE. It will also demonstrate to you the ability of Python scripts to perform just about any task you can perform manually in the GUI.

1.2 Using a script to define materials

When running simulations specific to your field of study you may find yourself reusing the same set of materials on a regular basis. For instance, if you analyze and simulate mostly products made by your own company, and these contain a number of steel components, you will need to define the material steel and along with its properties using the materials editor every time you begin a new simulation. One way to save yourself the trouble of defining material properties every time is to write a script that will accomplish this task. The Example 1.1 demonstrates this process.

Example 2.1 – Defining materials and properties

Let's assume you often use Titanium, AISI 1005 Steel and Gold in your product. The density, Young's Modulus and Poisson's Ratio of each of these materials is listed the following tables.

2 A Taste of Scripting

Properties of Titanium

Property	Metric	English
Density	4.50 g/cc	0.163 lb/in^3
Modulus of Elasticity	116 GPa	16800 ksi
Poisson's Ratio	0.34	0.34

Properties of AISI 1005 Steel

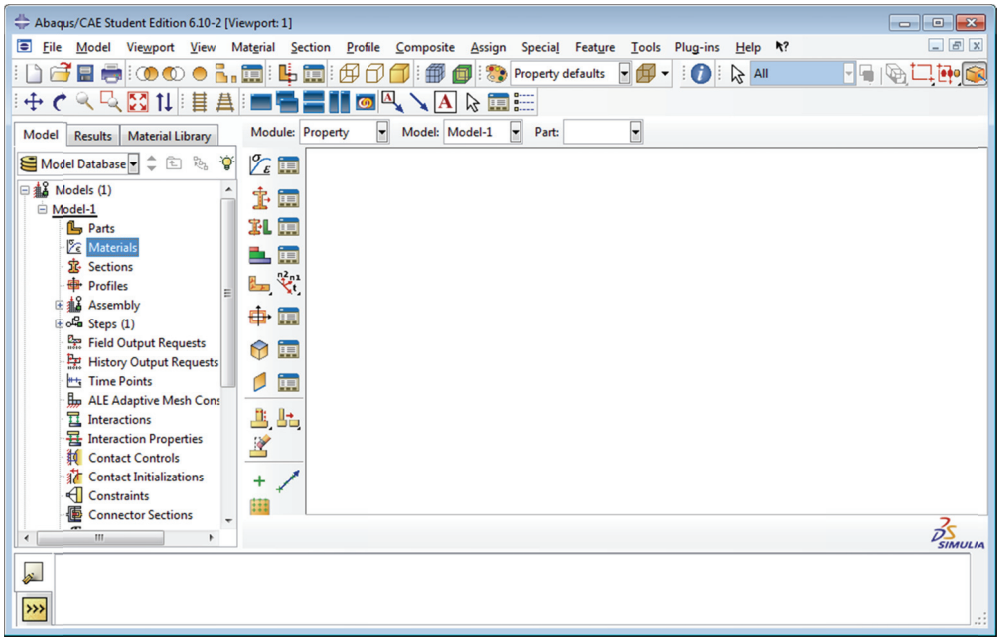
Property	Metric	English
Density	7.872 g/cc	0.2844 lb/in^3
Modulus of Elasticity	200 GPa	29000 ksi
Poisson's Ratio	0.29	0.29

Properties of Gold

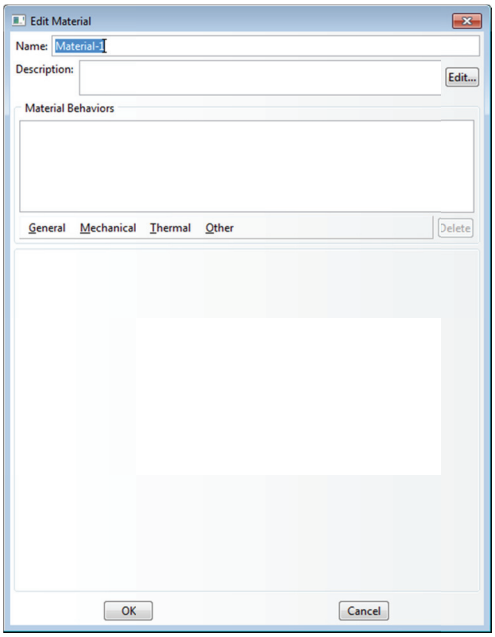
Property	Metric	English
Density	19.32 g/cc	0.6980 lb/in^3
Modulus of Elasticity	77.2 GPa	11200 ksi
Poisson's Ratio	0.42	0.42

Let's run through how you would usually define these materials in Abaqus CAE.

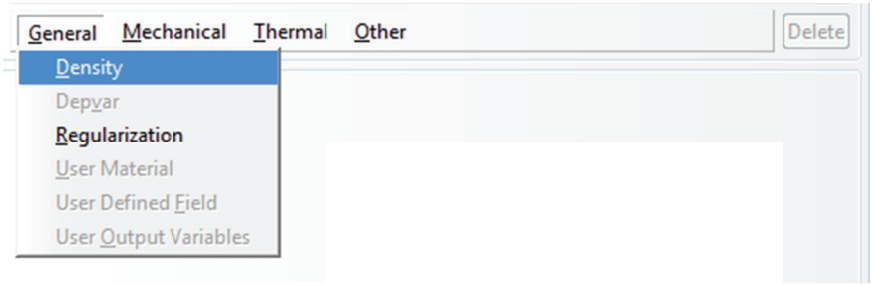
1. Startup Abaqus/CAE
2. If you aren't already in a new file click **File > New Model Database > With Standard/Explicit Model**
3. You see the model tree in the left pane with a default model called **Model-1**. There is no '+' sign next to the Materials item indicating that it is empty.



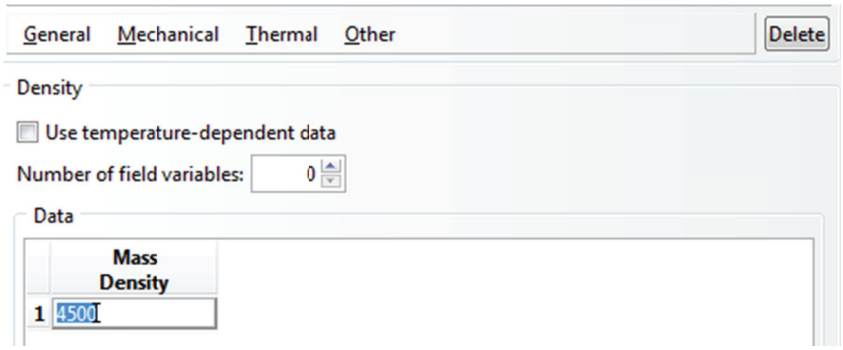
4. Double click the **Materials** item. You see the **Edit material** dialog box.



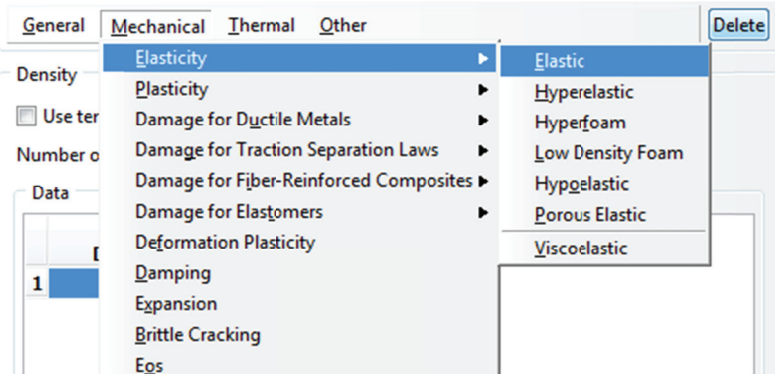
5. Name the material **Titanium**
6. Click **General > Density**.



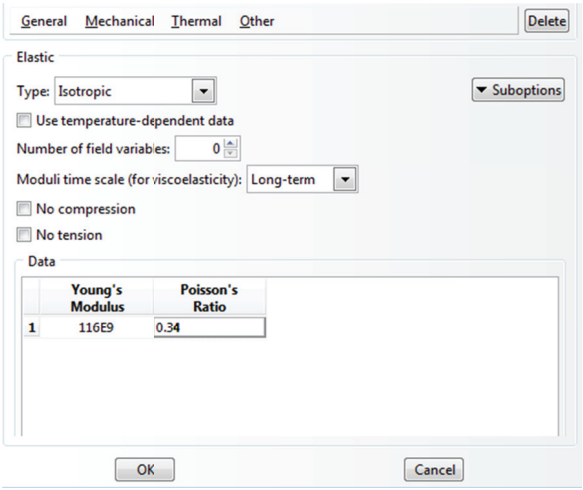
7. Let's use SI units with MKS (m, kg, s). We write the density of 4.50 g/cc as 4500 kg/m^3. Type this in as shown in the figure.



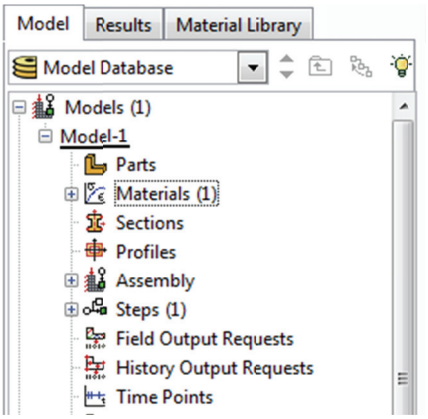
8. Then click **Mechanical > Elasticity > Elastic**



9. Type in the modulus of elasticity and Poisson's ratio. The Young's modulus of 116 GPa needs to be written as **116E9** Pa (or 116E9 N/m^2) to keep the units consistent. The Poisson's ratio of **0.34** remains unchanged.

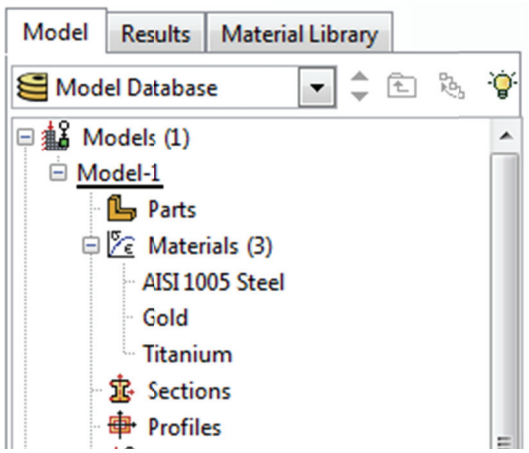


10. Click **OK**. The material is created and the model tree on the left indicates the presence of 1 material with the number **1** in parenthesis. Clicking the '+' symbol next to it reveals the name of the material **Titanium**, and double clicking it will reopen the **Edit material** window.



11. Repeat the process for the other 2 materials, **AISI 1005 Steel** and **Gold**. Remember to keep the units consistent with those used for **Titanium**.

12. When you're done the model tree should appear as it does in the figure with the 3 materials displayed.



That wasn't too hard. You defined 3 materials and you can now use these for the rest of your analysis. The problem is that you will need to define these materials in this manner all over again whenever you open a new file in Abaqus CAE to start a new study on your products. This is a tedious process, particularly if you have a lot of materials and you define a large number of their properties. Aside from consuming time there is also the chance of typing in a number wrong and introducing an error into your simulations, which will later be very hard to spot.

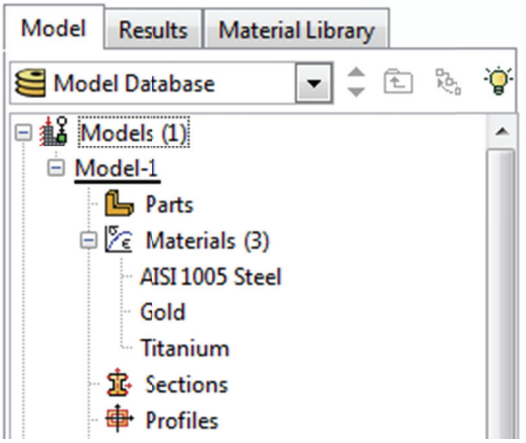
One way to fix this situation is to add your materials to the materials library. Then you could import the materials every time you created a new Abaqus file. Another way to do this would be in the form of a script. You type out the script once and place it in a file with the extension .py and every time you need these materials you go to **File > Run Script…**

Let's put a script together. Start by opening up a simple text editor. My personal favorite is Notepad++. It is free and it has got a clean interface. It also displays line numbers next to your code (making it easier to spot debugging errors) and can color code your script by auto-detecting Python from the file extension. On the other hand you may wish to use one of the Python editors from Python.org such as PythonWin. The idea is to create a simple text file, and then save it with a .py extension.

Open a new document in Notepad. Type in the following statements:

```
mdb.models['Model-1'].Material('Titanium')
mdb.models['Model-1'].materials['Titanium'].Density(table=((4500, ), ))
mdb.models['Model-1'].materials['Titanium'].Elastic(table=((200E9, 0.3), ))

mdb.models['Model-1'].Material('AISI 1005 Steel')
mdb.models['Model-1'].materials['AISI 1005 Steel'].Density(table=((7872, ), ))
mdb.models['Model-1'].materials['AISI 1005 Steel'].Elastic(table=((200E9, 0.29), ))

mdb.models['Model-1'].Material('Gold')
mdb.models['Model-1'].materials['Gold'].Density(table=((19320, ), ))
mdb.models['Model-1'].materials['Gold'].Elastic(table=((77.2E9, 0.42), ))
```

Save the file as 'ch1ex1.py'. Now open a new file in Abaqus CAE using **File > New**. Click on **File > Run Script...** The script will run, probably so fast you won't notice anything at first. But if you look closely at the **Materials** item in the model tree you will see the number 3 in parenthesis next to it indicating there are 3 defined materials. If you click the '+' sign you will see our 3 materials.



In fact if you double click on any of the materials, the **Edit Material** window will open showing you that the density and elastic material behaviors have been defined.

The script file has performed all the actions you usually execute manually in the GUI. It's created the 3 materials in turn and defined their densities, moduli of elasticity and Poisson's ratios. You could open a new Abaqus/CAE model and repeat the process of running the script and it would take about a second to create all 3 materials again.

If by chance you tried to decipher the script you just typed you may be a little lost. You see the words 'density' and 'elastic' as well as the names of materials buried within the code, so you can get a general idea of what the script is doing. But the rest of the syntax isn't too clear just yet. Don't worry, we'll get into the details in subsequent chapters.

1.3 To script or not to script..

Is writing a script better than simply storing the materials in the materials library? Well for one, it allows you to view all the materials and their properties in a text file rather than browsing through the materials in the GUI and opening multiple windows to view each property. Secondly you can make two or three script files, one for each type of simulation your routinely perform, and importing all the required materials will be as easy as **File > Run Script**. On the other hand if you store the materials in a material library you will need to search through it and pick out the materials you wish to use for that simulation each time.

At the end of the day it is a judgment call, and for an application as simple as this either method works just fine. But the purpose of this Example 1.1 was to demonstrate the power of scripting, and give you a feel for what is possible. Once you've read through the rest of the book and are good at scripting, you can make your own decision about whether a simulation should be performed with the help of a script or not.

1.4 Running a complete analysis through a script

You've seen how a script can accomplish a simple task such as defining material properties. A script however is not limited to performing single actions, you can in fact run your entire analysis using a script without having to open up Abaqus/CAE and see the GUI at all. This means you have the ability to create parts, apply material properties, assign sections, apply loads and constraints, define sets and surfaces, define interactions and constraints, mesh and run the simulations, and also process the results, all through a script. In the next example you will write a script that can do all of these things.

Example 2.2 – Loaded cantilever beam

Just as in the previous example, we will once again begin with demonstrating the process in Abaqus/CAE and then perform the same tasks with a script. We're going to create a simple cantilever beam 5 meters long with a square cross section of side 0.2 m made of AISI 1005 Steel. Being a cantilever this beam will be clamped at one end. That means that it can neither translate along the X, Y or Z axes, nor can it rotate about them at that

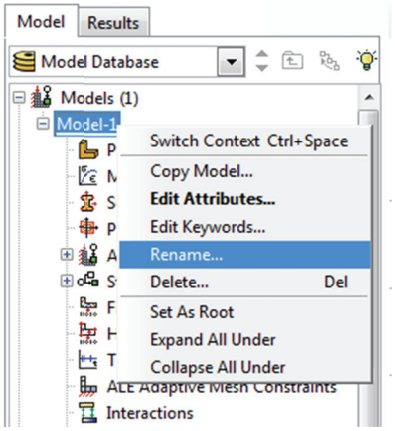
fixed end. This is also known as an encastre condition. A pressure load of 10 Pa will cause the beam to bend downwards with the maximum deflection experienced the free end.

Field output and history output data will be collected. Field output data provides information on the state of the overall system during the load step, such as the stresses and strains. Instead of using the defaults, we will instruct Abaqus to track the stress components and invariants, total strain components, plastic strain magnitude, translations and rotations, reaction forces and moments, and concentrated forces and moments. History output data provides information on the state of a smaller section such as a node at frequent intervals. For this we will allow Abaqus to track the default variables for history output.

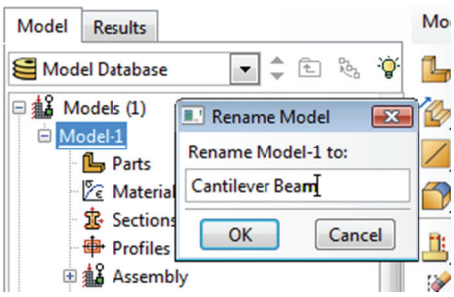
We will mesh the beam using an 8-node linear brick, reduced integration element (C3D8R) with a mesh size of 0.2. We will create a job, submit it, and inspect the results.

Let's start by performing these tasks in the GUI mode using Abaqus CAE.

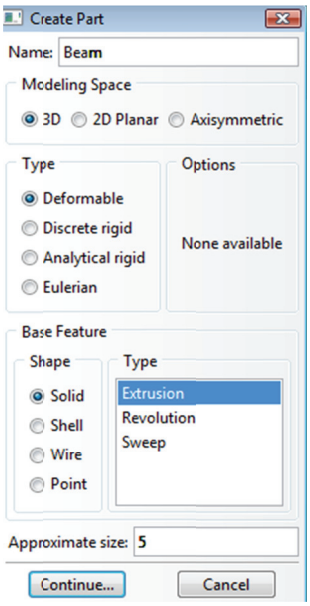
1. Startup Abaqus/CAE
2. If you aren't already in a new file click **File > New**
3. In the Model Database panel right click **Model-1** and choose **Rename….**



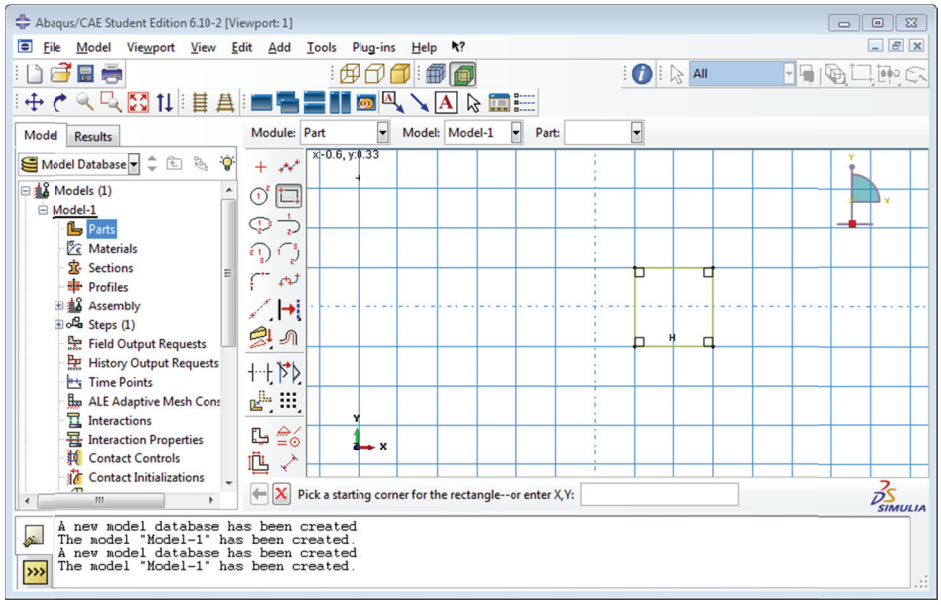
4. Type in **Cantilever Beam**. **Model-1** will change to **Cantilever Beam** in the tree.



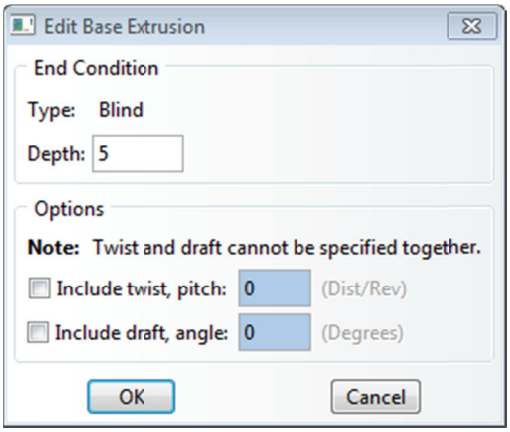
5. Double click on the **Parts** item. The **Create Part** dialog is displayed. Name the part **Beam**. In the **Modeling Space** section, choose **3D**. For the **Type** choose **Deformable**. For **Base Feature** choose **Solid** as the shape and **Extrusion** as the type. Set the **Approximate Size** to **5**. Press **Continue..**



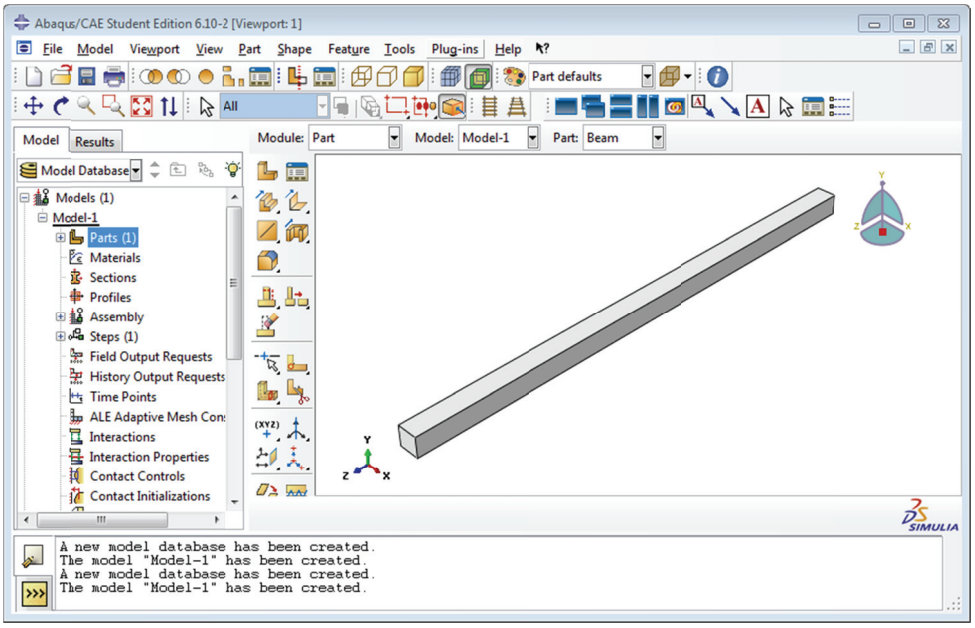
6. You find yourself in the **Sketcher** window. Select the rectangle tool from the toolbar. For the first point click on (0.1, 0.1). For the second point click on (0.3, -0.1). A rectangle is drawn with these two points as the vertices.



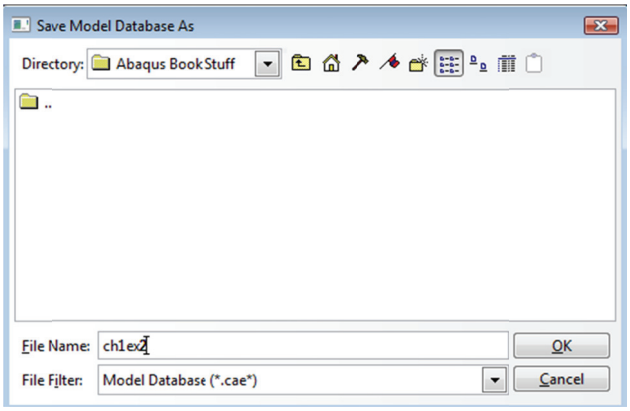
7. Click the red **X** button at the bottom of the window indicating **End procedure** and then click **Done**.
8. In the **Edit Base Extrusion** window set **Depth** to **5**.



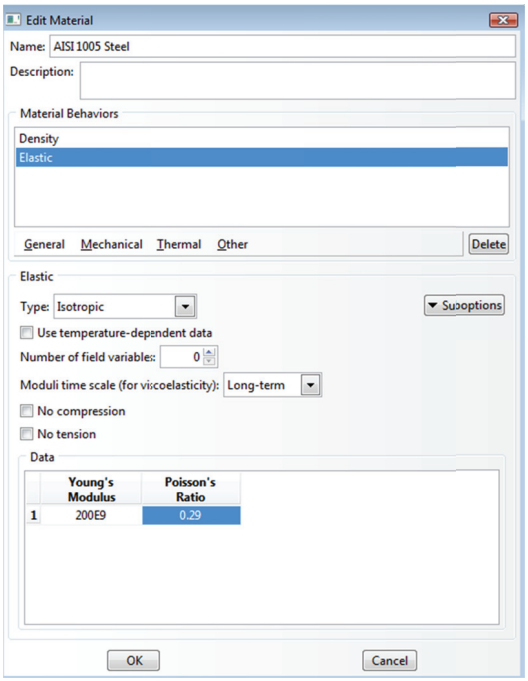
9. Click **OK**. You will see a 3D rendering of the part **Beam** you just made. The **Parts** item in the model tree now has a sub-item called **Beam**.



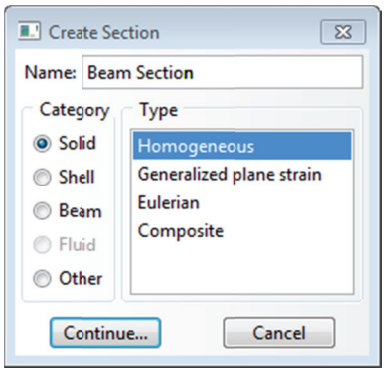
10. Now would be a good time to save your file. Choose **File > Save**. Select the directory you save your files in and name this file 'cantilever_beam.cae' (or something more creative if you prefer)



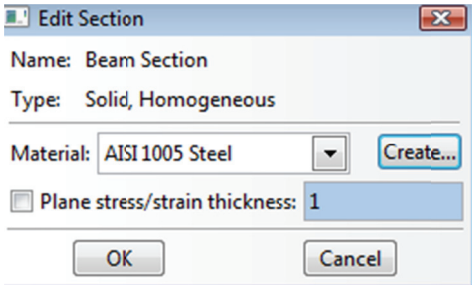
11. Double click the **Materials** item in the model tree. Name it **AISI 1005 Steel**. Set **General > Density** to **7872** kg/m^3. Set **Mechanical > Elasticity > Elastic** to a Young's Modulus of **200E9** N/m^2 and a Poisson's Ratio of 0.**29**.



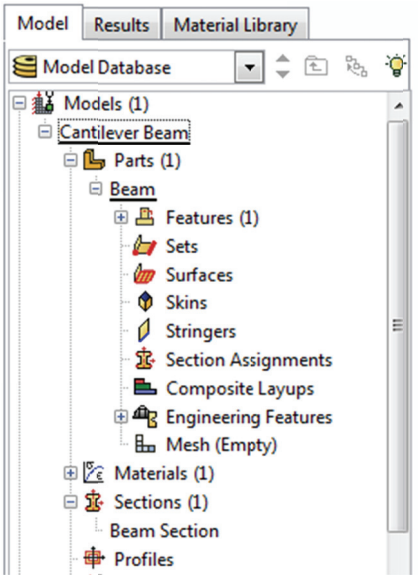
12. Click **OK**. The material is added to the model tree.
13. Double click on the **Sections** item. The **Create Section** window is displayed. Name it **Beam Section**. Set the **Category** to **Solid** and the **Type** to **Homogeneous** if this isn't already the default.



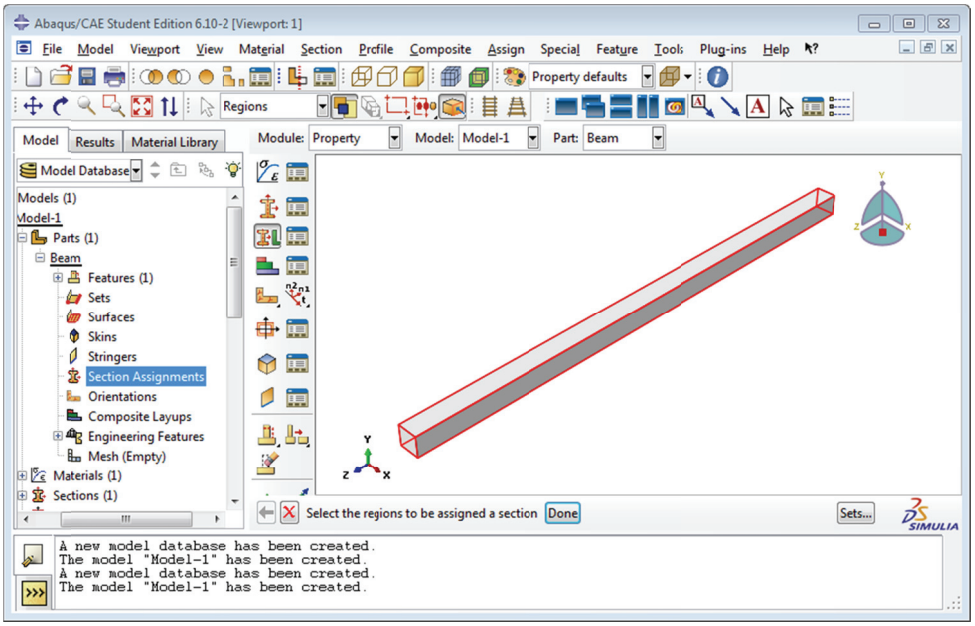
14. Click **Continue**. The **Edit Section** window is displayed with the **Name** set to **Beam Section** and **Type** set to **Solid, Homogeneous**. Under the **Material** drop down menu choose **AISI 1005 Steel** which is the material you created a moment ago.



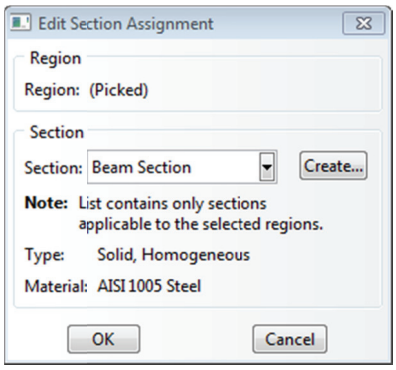
15. Click **OK**. You will notice that the **Sections** item in the model tree now has a sub-item called **Beam Section**.
16. Next we need to assign this section to the part **Beam**. Expand the **Parts (1)** item by clicking the + symbol next to it to reveal the **Beam** item. Expand that too to reveal a number of sub-items such as **Features**, **Sets**, **Surfaces** and so on.



17. Double click the sub-item **Section Assignments**. You will see the hint **Select the regions to be assigned a section** below the viewport. Hover your mouse over the beam in the viewport and when all its edges light up click to select it.

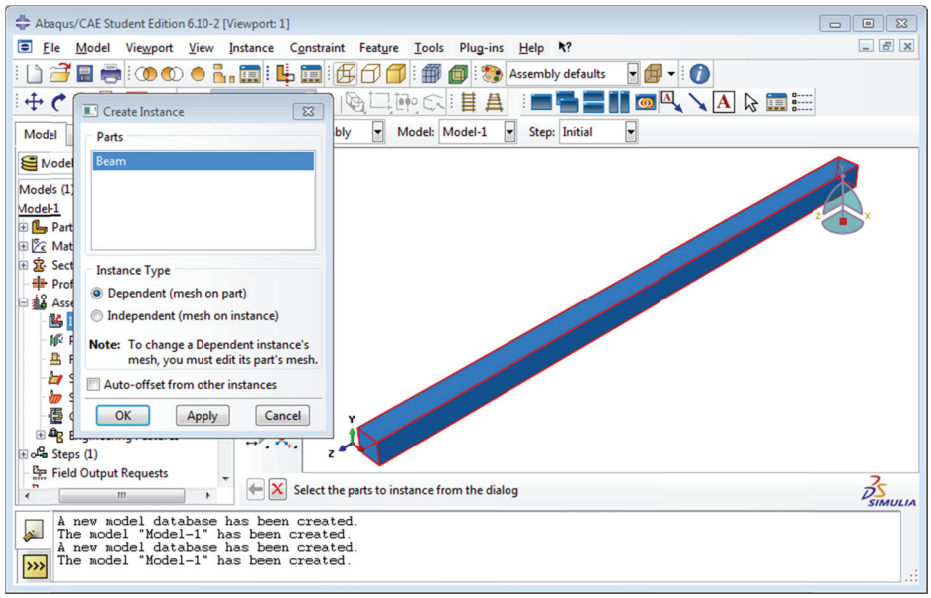


18. Click **Done**. You see the **Edit Section Assignment** window. Set the **Section** to **Beam Section** which is the section you created in steps 13-15.

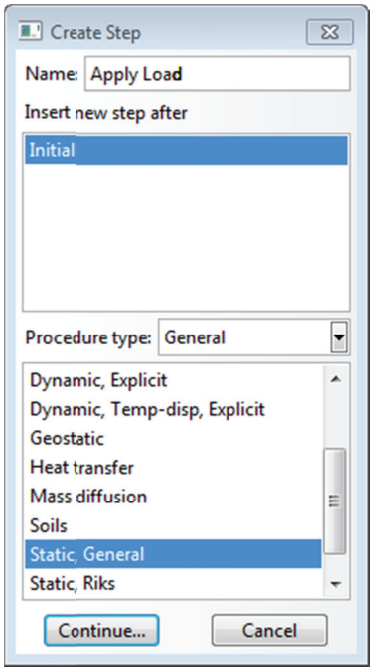


19. Click **OK**. The **Section Assignments** item now has 1 sub-item **Beam Section (Solid, Homogeneous).** The part in the viewport changes color (to green on my system) indicating it has been assigned a section.
20. Let's import the part into an assembly. Click the + symbol next to the **Assembly** item in the model tree and double-click the **Instances** sub-item. You see the **Create**

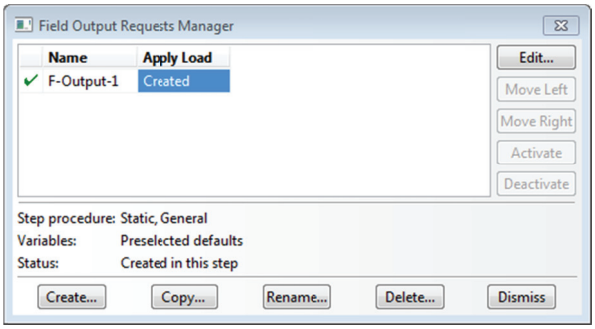
Instance window. For **Parts**, **Beam** is the only option available and it is selected by default. For the **Instance Type** choose **Dependent (mesh on part).**



21. Click **OK**. The **Instances** sub-item of the **Assembly** item now has a sub-item of its own called **Beam-1**. You can right-click on it and choose **Rename….** Change the name to **Beam Instance**.
22. Next we create a step in which to apply the load. Notice that the **Steps** item in the model tree already has the **Initial** step. Double-click the **Steps** item. The **Create Step** window is displayed. Name the step **Apply Load**. For **Insert new step after** the only option is **Initial** and it is selected by default. Set the **Procedure type** to **General** from the drop down menu. In the list scroll down till you see **Static, General** and select it.

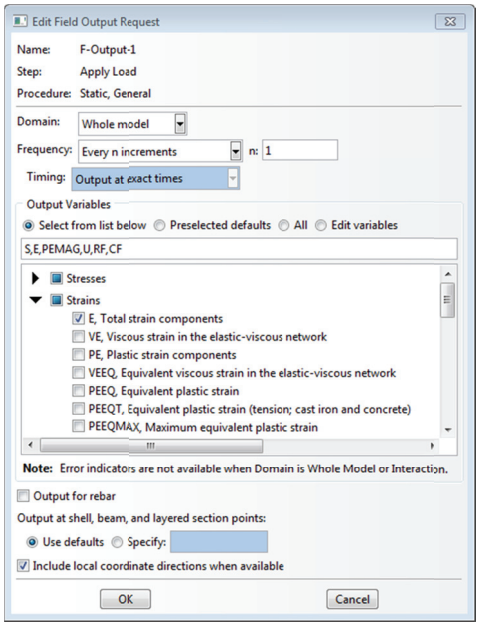


23. Click **Continue…**. You see the **Edit Step** window. For the description type in **Load is applied during this step**. Leave everything else set to the defaults.
24. Click **OK**. You'll notice that the **Steps** item in the Model Database now has 2 steps, **Initial** and **Apply Load**.
25. Let's now create the field output requests. Right click the **Field Output Requests** item in the model tree and choose **Manager**. You see the **Field Output Requests Manager** window with an output request **F-Output-1** created in the **Apply Load** step.

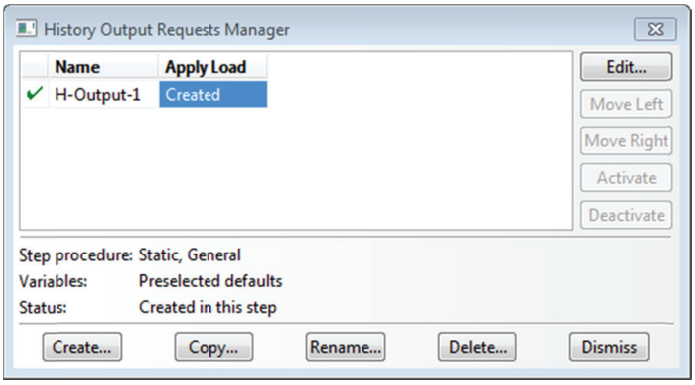


Click the **Edit** button. You notice a number of output variables selected by default. On top of the list of available output variables you see a comma separated listing of the ones selected which by default reads **CDISP, CF, CSTRESS, LE, PE, PEEQ, PEMAG, RF, S, U,.**

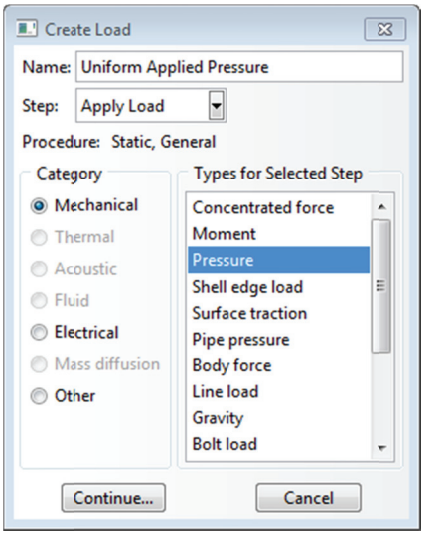
26. From the **Strains** remove **PE, Plastic strain components**, **PEEQ, Equivalent plastic strain** and **LE, Logarithmic strain components**. Add **E, Total strain components**. Remove **Contact** entirely. The variables you are left with are displayed above as **S,E,PEMAG,U,RF,CF**



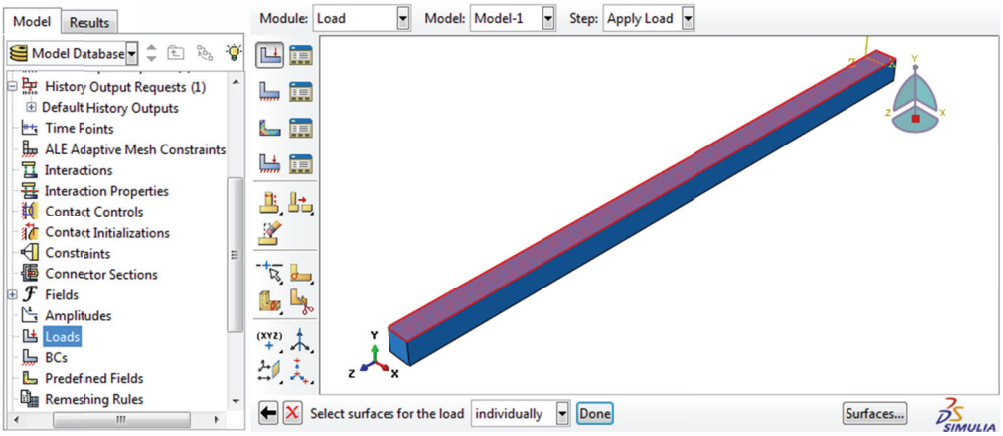
27. Click **OK**. Then click **Dismiss…** to close the **Field Output Request Manager** window. In the model tree right click the **F-Output-1** sub-item of **Field Output Requests** and rename it **Selected Field Outputs**.
28. Let's move on to history output requests. Right click **History Output Requests** in the model tree and choose **Manager**. You see the **History Output Requests Manager** window. It is very similar to the **Field Output Requests Manager** window.



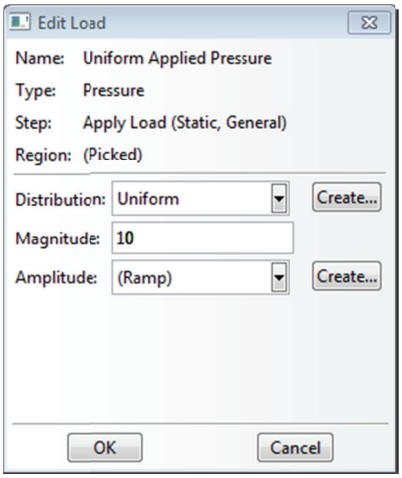
29. If you click **Edit** you can see the variables selected by default. We're going to leave the default variables selected so **Cancel** out of the **Edit History Output Requests** window. In the model tree right click the **H-Output-1** sub-item of **History Output Requests** and rename it **Default History Outputs**.
30. It's time to apply loads to the beam. In the model tree double click the **Loads** item. You see the **Create Load** window. Name the load **Uniform Applied Pressure**. For the step select **Apply Load**. Under **Category** choose **Mechanical**. And from the **Types for Selected Step** list choose **Pressure**.



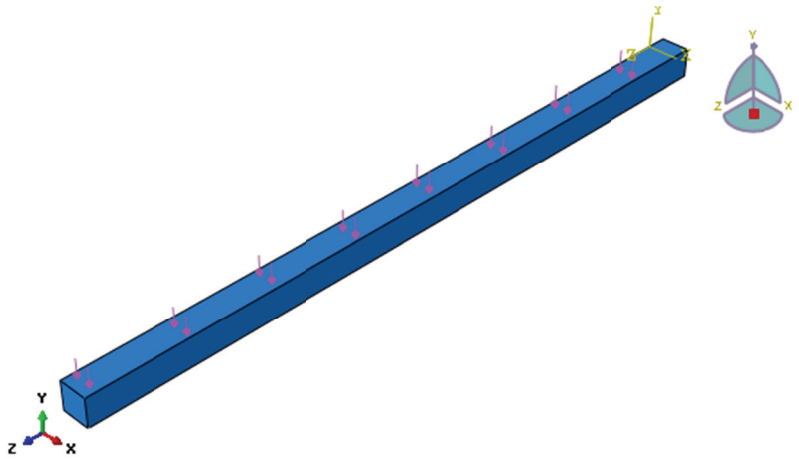
31. Click **Continue….** The viewport displays a hint at the bottom **Select surfaces for the load**. Hover your mouse over the top surface of the beam till its edges light up. Click to select.



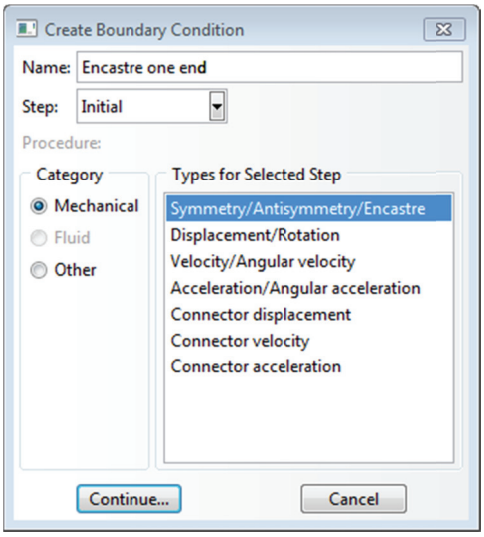
32. Click **Done**. You see the **Edit Load** window. For **Distribution** choose **Uniform** from the drop down list. For **Magnitude** enter a value of **10** Pa (just type in 10 without units).



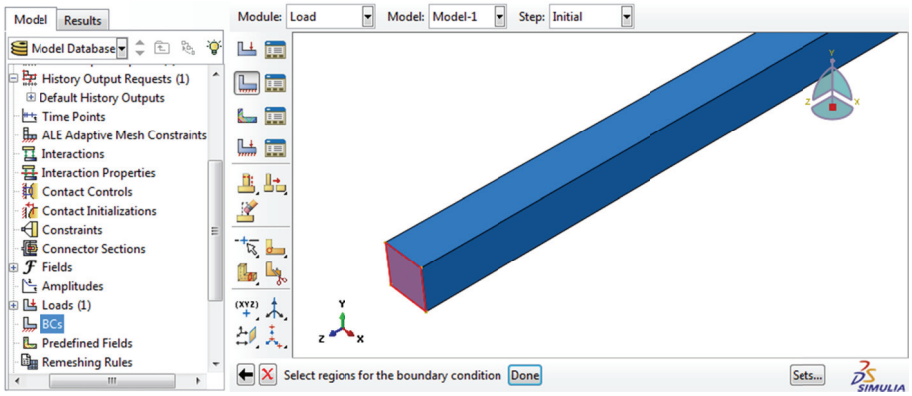
33. Click **OK**. The viewport updates to show the pressure being applied on the top surface with the arrows representing the direction. Also the **Loads** item in the Model Database tree now has a sub-item called **Uniform Applied Pressure**.



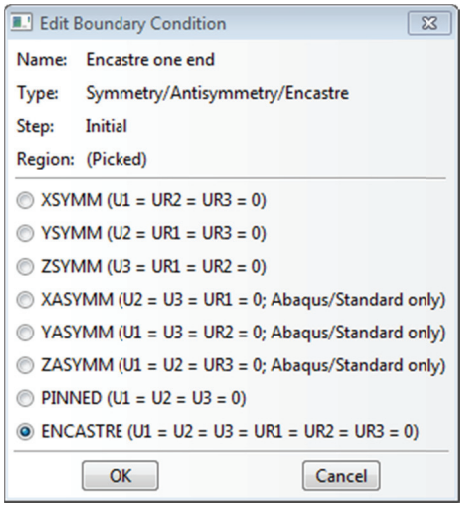
34. The next step is to apply the boundary conditions or constraints. Double click on the **BCs** item in the Model Database tree. You see the **Create Boundary Condition** window. Name it **Encastre one end**. Change **Step** to **Initial**. Under **Category** choose **Mechanical**. From the available options for **Types for Selected Step** choose **Symmetry/Antisymmetry/Encastre**.



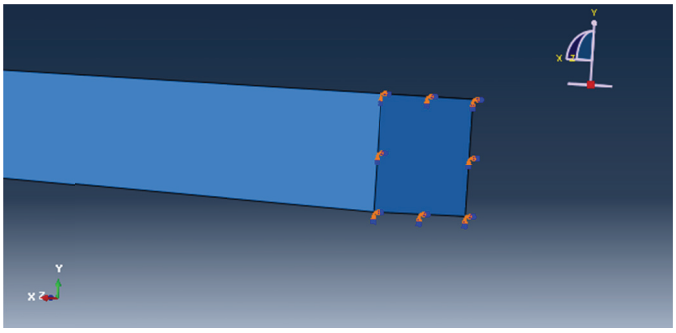
35. Click **Continue….** The viewport displays a hint at the bottom **Select regions for the boundary condition**. Hover your mouse over the surface at one end of the beam till its edges light up. Click to select it.



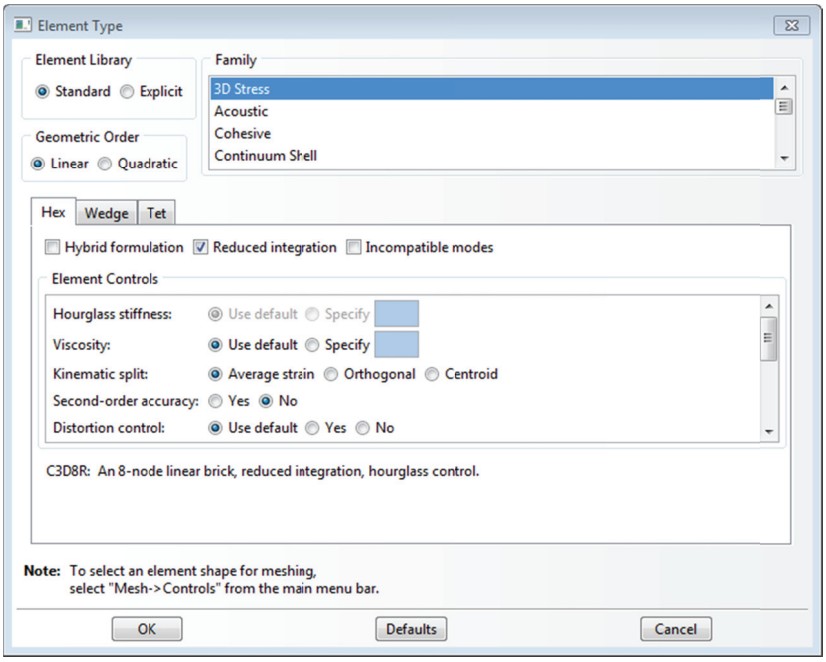
36. Click **Done**. You see the **Edit Boundary Condition** window. Choose **ENCASTRE (U1 = U2 = U3 = UR1 = UR2 = UR3 =0)**. This will clamp the beam at this end.



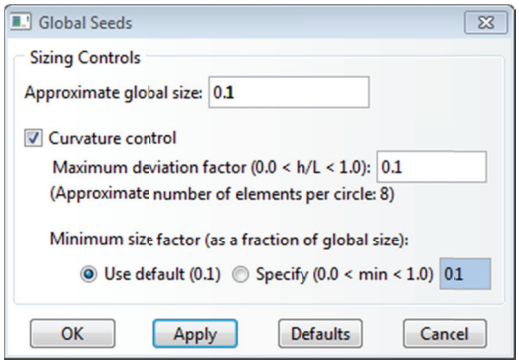
37. The viewport will update to show the end of the beam being clamped. Also the **BCs** item now has a sub-item called **Encastre one end**.



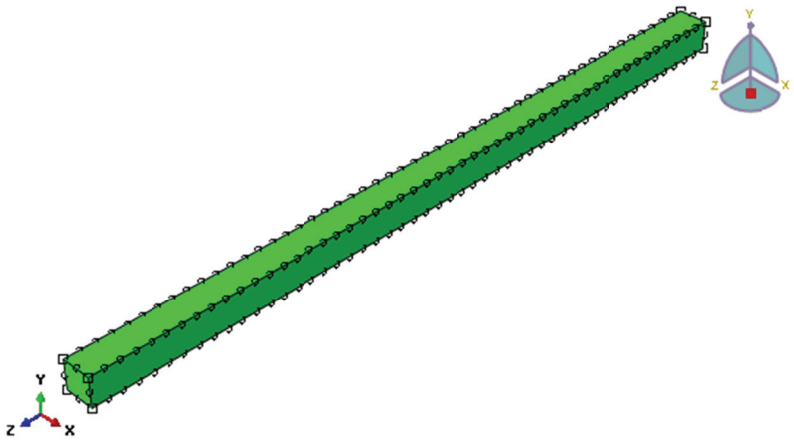
38. If you haven't been saving your work all along now would be a good time to do it. We're going to mesh the part and then run the simulation.
39. In the model tree expand the **Parts** item again, and then the **Beam** sub-item. You see the **Mesh (Empty)** sub-item at the bottom. Double-click it. You are now in mesh mode and you notice the toolbar next to the viewport changes to provide you with mesh tools.
40. Using the menu bar go to **Mesh > Element Type**. The **Element Type** window is displayed. For **Element Library** choose **Standard**, for **Geometric Order** choose **Linear**, and for **Family** choose **3D Stress** from the list. Leave everything else at the defaults. You will notice the description **C3D8R: An 8-node linear brick, reduced integration, hourglass control** near the bottom of the window.



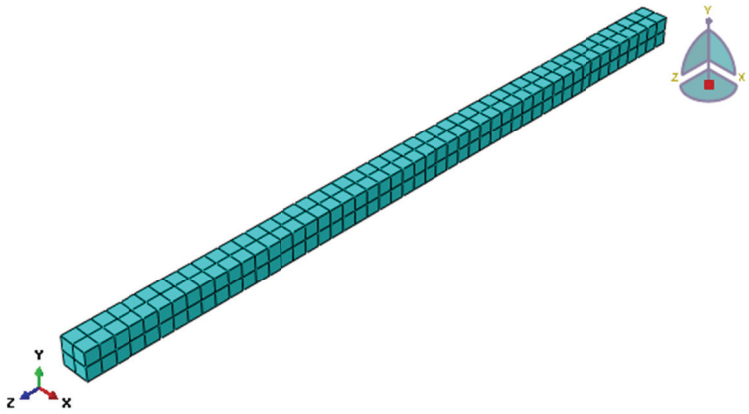
41. Click **OK**.
42. Then use the menu bar to navigate to **Seed > Part**. The **Global Seeds** window is displayed. Change the **Approximate global size** to **0.2**, which is the width of our beam. Set the **Maximum deviation factor** to **0.1**.



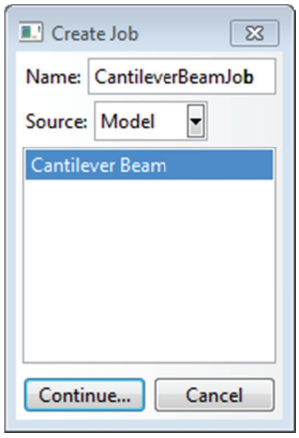
43. The beam in the viewport updates to show where the nodes have been applied.



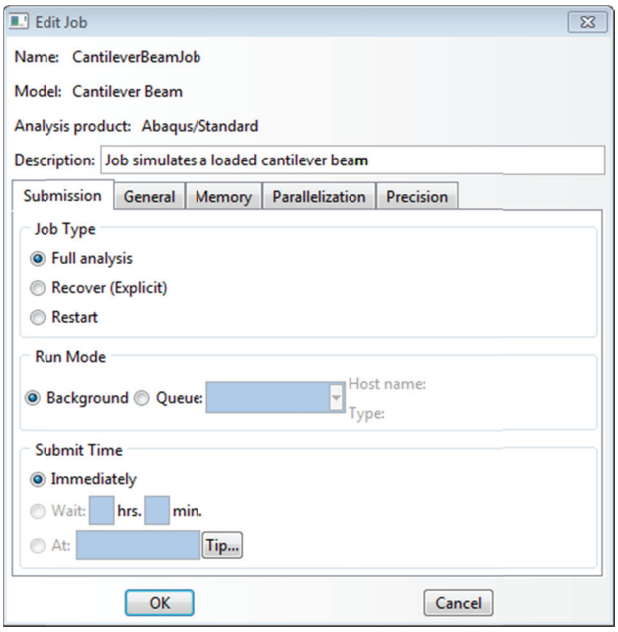
44. Then from the menu bar go to **Mesh > Part**. You see the question **OK to mesh the part?** at the bottom of the viewport window. Click on **Yes**. The part is meshed. The **Mesh** item in the model tree no longer has the words **(Empty)** next to it.



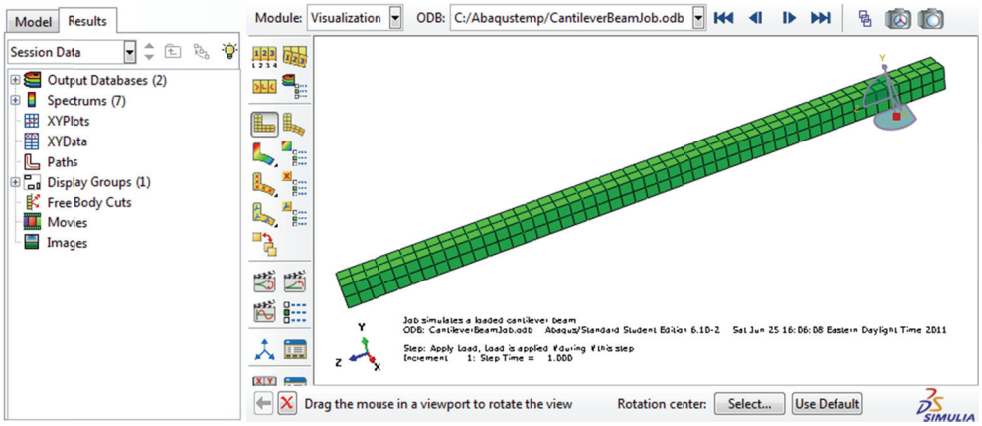
45. Now it is time to create the analysis job.
46. All the way at the bottom of the model tree you see **Analysis** with the sub-item **Jobs**. Double-click on it. The **Create Job** window is displayed. Name it **CantileverBeamJob**. Notice that there are no spaces in the name. Putting a space in the job name can cause problems because Abaqus uses the job name as part of the name of some of the output files such as the output database (.odb) file. **Source** is set to **Model** and the only model you can select from the list is **Cantilever Beam**.



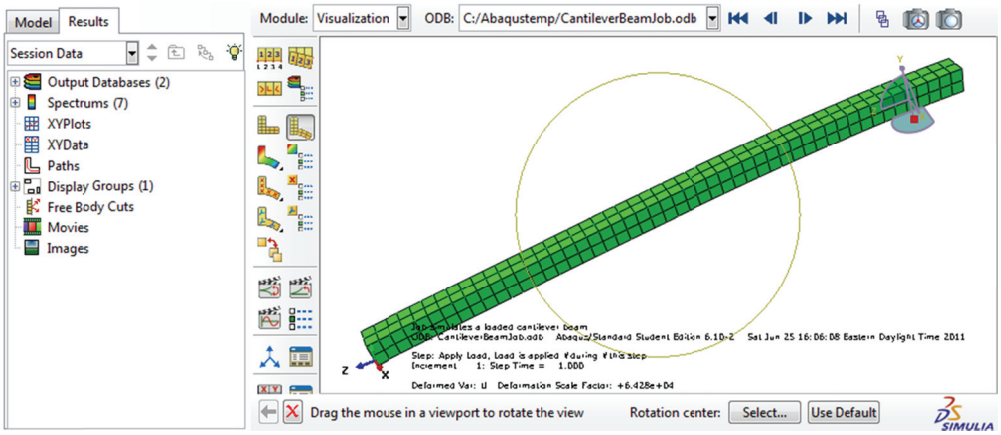
47. Click **Continue….** You see the **Edit Job** window. In the **Description** textbox type in **Job simulates a loaded cantilever beam**. Set the **Job Type** to **Full Analysis**. Leave the other settings to default. Notice that in the **Memory** tab there is an option for **Memory allocation units**. On my system the option selected is **Percent of physical memory**, and for the **Maximum preprocessor and analysis memory** my system defaults to **50%**. You might wish to play with these numbers if your computer has insufficient resources.



48. Notice that the **Jobs** item in the model tree now has **CantileverBeamJob** listed (you might have to hit the '+' symbol to see it). Right-click on it and choose **Submit**.
49. The job starts running. You see the words **(Submitted)** appear next to its name in parentheses, then a few seconds later you see **(Running)** and when the simulation is complete you see **(Completed)**.
50. Right click on **CantileverBeamJob (Completed)** and choose **Results**. You see the undeformed shape.



51. Click the **Plot Deformed Shape** button in the toolbar to the left of the viewport. You will see your deformed beam. Of course the deformation has been exaggerated by Abaqus. You can change that if you wish by going to **Options > Common…** if you wish.



You have created and run a complete simulation in Abaqus/CAE. It was a very basic setup, but it covered all the essentials from creating a part and assigning sections and material properties to applying loads and constraints and meshing. Now we're going to watch a script perform all the same actions that we just did.

Open up a text editor such as Notepad++ and type in the following script.

```
# ***************************************************************************
# Cantilever Beam bending under the action of a uniform pressure load

# ***************************************************************************

from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ----------------------------------------------------------------------
# Create the model
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# ----------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))

# b) Create a 3D deformable part named "Beam" by extruding the sketch
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)

# ----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs
# modulus and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

```
# -------------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the beam
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')

# Assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')

# -----------------------------------------------------
# Create the assembly

                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Create the step

                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Create the field output request


                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Create the history output request


                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Apply pressure load to top surface


                          (Statements removed from preview)
```

```
# ------------------------------------------------------------------------
# Apply encastre (fixed) boundary condition to one end to make it cantilever

                          (Statements removed from preview)


# ------------------------------------------------------------------------
# Create the mesh

                          (Statements removed from preview)


# ------------------------------------------------------------------------
# Create and run the job

                          (Statements removed from preview)


# ------------------------------------------------------------------------
# Post processing

import visualization

beam_viewport = session.Viewport(name='Beam Results Viewport')
beam_Odb_Path = 'CantileverBeamJob.odb'
an_odb_object = session.openOdb(name=beam_Odb_Path)
beam_viewport.setValues(displayedObject=an_odb_object)
```

Typing out the above code might be a real pain and you'll likely mistype a few variable names or make other syntax errors creating a lot of bugs. It might be a better idea just to use the source code provided with the book – cantilever_beam.py.

Open a new Abaqus model. Then go to **File > Run Script**. The script will recreate everything you did manually in Abaqus/CAE. It will also create and submit the job so you will probably notice the analysis running for a few seconds after you run the script. You can then right click on the 'CantileverBeamJob' item in the model tree and choose

Results to see the output. It will be identical to what you got when performing the simulation in the GUI.

1.5 Conclusion

In the example we did not use the script to accomplish anything that could not be done in Abaqus/CAE. In fact we first performed the procedure in Abaqus/CAE before writing our script. But I wanted to drive home an important point: You can do just about anything in a script that you can do in the GUI. Once you're able to script a basic simulation, you'll be able to move on to more complex tasks that would really only be feasible with a script such as making automated decisions when creating the simulation or performing repetitive actions within the study.

As for the script from this example, we're going to take a closer at it in Chapter 4. Before we can do this you're going to have to learn a little Python syntax in Chapter 3. But first let's take a look at the different ways of running a script in Chapter 2.

2

Running a Script

2.1 Introduction

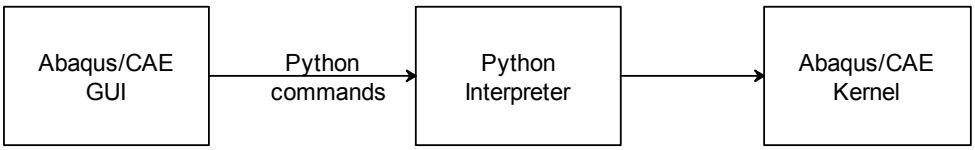
This chapter will help you understand how Python scripting fits into Abaqus, and also point out some of the different ways a script can be run. While you may choose to use only one of the methods available, it is handy to know your options.

2.2 How Python fits in

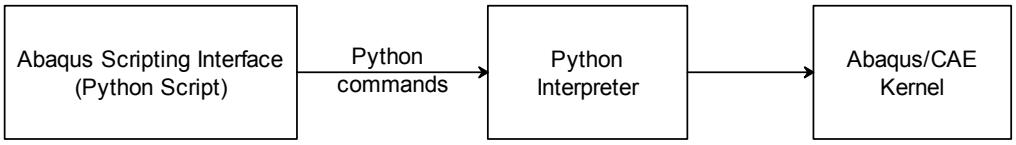
A few years ago Abaqus existed purely as a finite element solver. It had no preprocessor or postprocessor. You created text based input files (.inp), submitted them to the solver using the command line, and got text based output files. Today it has a preprocessor which generates the input file for you – Abaqus/CAE (CAE stands for Complete Abaqus Environment), and a postprocessor that helps you visualize the results from the output database – Abaqus/Viewer. When you use Abaqus/CAE, the actions you perform in the GUI (graphical user interface) generate commands in Python, and these Python commands are interpreted by the Python Interpreter and sent to the Abaqus/CAE kernel which executes them. For example when you create a new material in Abaqus/CAE, you type in a material name and specify a number of material behaviors in the 'Edit Material' dialog box using the available menus and options. When you click OK after this, Abaqus/CAE generates a command or a number of commands based on what you have entered and sends it to the kernel. They may look something like:

```
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

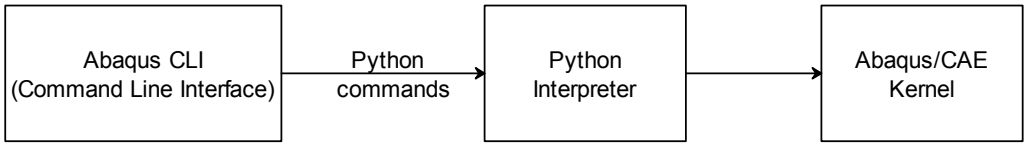
In short, the Abaqus/CAE GUI is the easy-to-use interface between you, the user, and the kernel, and the GUI and kernel communicate using Python commands.



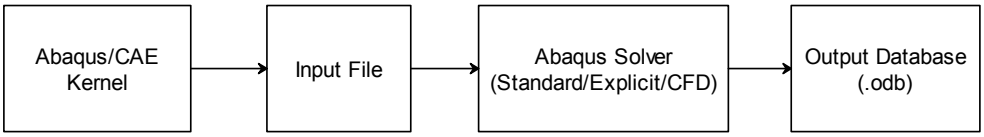
The Abaqus Scripting Interface is an alternative to using the Abaqus/CAE GUI. It allows you to write a Python script in a .py file and submit that to the Abaqus/CAE Kernel.



A third option is to type scripts into the kernel command line interface (CLI) at the bottom of the Abaqus/CAE window.



The Abaqus/CAE kernel understands the model and creates an input file that can be submitted to the solver. The solver accepts this input file, runs the simulation, and writes its output to an output database (.odb) file.



2.3 Running a script within Abaqus/CAE

You have the option of running a script from within Abaqus/CAE using the **File > Run Script…** menu option. You can do this if your script irrespective of whether your script only performs a single task or runs the entire simulation.

2.3.1 Running a script in GUI to execute a single or multiple tasks

If you have a script that performs a single independent task or multiple tasks assuming some amount of model setup has already been completed or will be performed later, you need to run that script in Abaqus/CAE. For instance, in Example 1.1 of Chapter 1, we wrote a script which only creates materials. On its own this script cannot run a simulation, it does not create a part, assembly, steps, loads and boundary conditions and so on. However it can be run within Abaqus/CAE to accomplish a specific task. When we ran the script using **File > Run Script…** you noticed the model tree get populated with new materials. You could then continue working on the model after that.

Such scripts will not run as standalone from the command line, or at least they won't accomplish anything.

2.3.2 Running a script in GUI to execute an entire simulation

If you have a script that can run the entire simulation, from creating the part and materials to applying loads and constraints to meshing and running the job, one way to run it is through the GUI using *File > Run…* This was demonstrated in Example 1.2 of Chapter 1. However such a script can also be run directly from the command line.

2.4 Running a script from the command line

In order to run a script from the command line, the Abaqus executable must be in your system path.

Path

The path is a list of directories which the command interpreter searches for an executable file that matches the command you have given it. It is one of the environment variables on a Windows machine.

The directory you need to add to your path is the "Commands" directory of your Abaqus installation. By default Abaqus Student Edition v6.10 installs itself to directory "C:\SIMULIA\Abaqus". It likely did the same on your computer unless you chose to install it to a different location during the installation procedure. One of the sub-directories of "C:\SIMULIA\Abaqus" is "Commands", so its location is "C:\SIMULIA\Abaqus\Commands". This location needs to be added to the system path.

Check if Abaqus is already in the path

The first thing to do is to check if this location has already been added to your path as part of the installation. You can do this by opening a command prompt. To access the command prompt in Windows Vista or Windows 7, click the Start button at the lower left corner of your screen, and in the 'Start search' bar that appears right above it type 'cmd' (without the quotes) and hit enter. In Windows XP you click the Start button, click 'Run', and type in 'cmd' and click OK. You will see your blinking cursor. Type the word 'path' (without the quotes). You wil`l see a list of directories separated by semicolons that are in the system path. If Abaqus has been add to the path, you will see "C:\SIMULIA\Abaqus\Commands" (or wherever your Commands folder is) listed among the directories. If not, you need to add it manually to the path.

Add Abaqus to the Path

Adding a directory to the path differs slightly for each version of Windows. There are many resources on the internet that instruct you on how to add a variable to the path and a quick Google search will reveal these. As an example, this is how you add Abaqus to the path in Windows Vista and Windows 7.

1. Right click **My Computer** and choose **Properties**
2. Click **Advanced System Settings** in the menu on the left.
3. In the System Properties window that opens, go to the **Advanced** tab. At the bottom of the window you see a button labeled **Environment Variables..**. Click it.
4. The environment variables window opens. In the **System variables** list, scroll down till you see the **Path** variable. Click it, then click the **Edit** button. You see the **Edit System Variable** window.
5. The variable name shall be left at its default of **Path**. The variable value needs to be modified. It contains a number of directories, each separated by a semi colon. It may look something like **C:\Windows\System32\;C:\Windows\;C:\Windows\System32\Wbem**. At the end of it, add another semi colon, and then type in **C:\SIMULIA\Abaqus\Commands**. So it should now look something like **C:\Windows\System32\;C:\Windows\;C:\Windows\System32\Wbem;C:\SIMULIA\Abaqus\Commands**. Click **OK** to close the window, and click **OK** to close the **Environment Variables** window.

6. Now if you go back to the command prompt and type **path**, you see the path has been updated to include Abaqus

Running the script from the command line

Now that Abaqus is in the system path, you can run your scripts from the command line.

First you navigate to the folder containing your script files using DOS commands such as **cd** (change directory) command. For example, when you start the command prompt, if your cursor looks something like **C:\Users\Gautam>,** and your script is located in the folder **C:\Users\Gautam \Desktop\Abaqus Book**, then type in

```
cd C:\Users\Gautam \Desktop\Abaqus Book
```

and press Enter. Your cursor will now change to **C:\Users\Gautam\Desktop\Abaqus Book>**

You are now in a position to run the script with or without the Abaqus/CAE GUI being displayed.

2.4.1 Run the script from the command line without the GUI

Type the command to run the script without the Abaqus/CAE GUI. The exact command varies depending on the version of Abaqus.

In the commercial version of Abaqus you would type

```
abaqus cae noGUI= "cantilever_beam.py"
```

In the student edition (SE) version 6.9-2 you would type

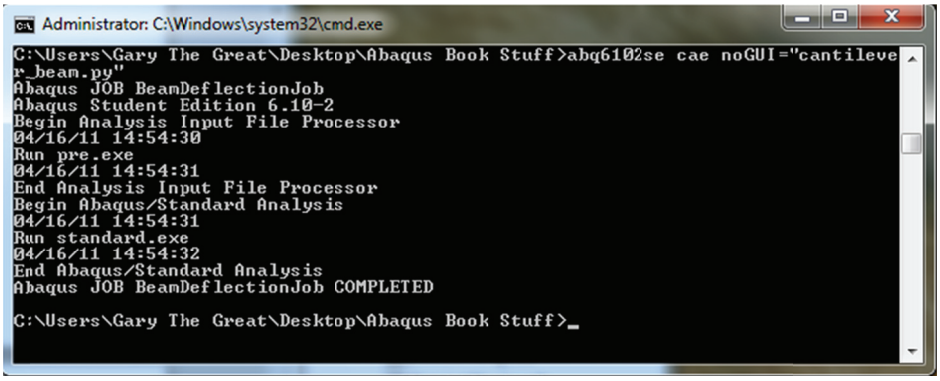
```
abq692se cae noGUI="cantilever_beam.py"
```

In the student edition (SE) version 6.10-2 you would type

```
abq6102se cae noGUI="cantilever_beam.py"
```

Notice the difference in the first word of all these statements. If you are not using either of these versions the command you use will be different as well. To figure out exactly what it is, go to the ‘Commands’ folder in the installation directory and look for a file with the extension ‘.bat’. In the commercial version of Abaqus this file is called ‘abaqus.bat’, hence in the commercial version you use the command ‘abaqus cae

noGUI="cantilever_beam.py". In Abaqus 6.10-2 student edition, the file is called 'abq6102se.bat' which is why the command 'abq6102se cae noGUI="cantilever_beam.py" has been used. Depending on the name of your file, change the first word in the statement.

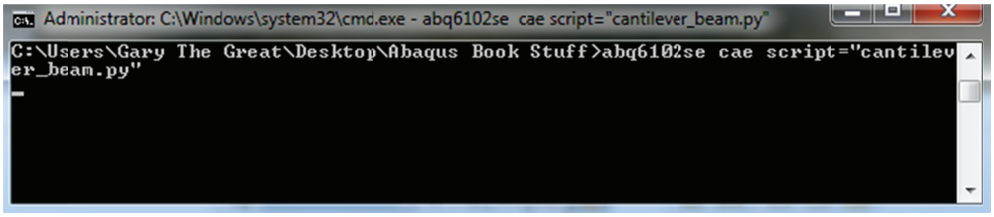


When you run your scripts in this manner, you will not see the GUI at all. While the script is running, you will notice that the cursor is busy and you cannot type in any other commands at the prompt. This is because we have used the built in method waitForCompletion() in the script which prevents the user from executing other DOS commands while the simulation is running. We will take a look at this statement again a little later, just be aware that if we did not include the waitForCompletion() command in the script, the prompt would continue to remain active even while the simulation is being run. And if you find yourself running batch files, or linking your simulations with optimization software such as ISight or ModelCenter, this knowledge will come in handy.

2.4.2 Run the script from the command line with the GUI

If on the other hand you wish to have the GUI displayed replace the word 'noGUI' with 'script'. So in student edition version 6.10-2 you would type

```
abq6102se cae script="cantilever_beam.py"
```



When you run your scripts in this manner, Abaqus/CAE will open up and the script is run within it. In addition the cursor will remain busy (as seen in the figure), and will only be released once you close that instance of Abaqus/CAE.

2.5 Running a script from the command line interface (CLI)

The kernel command line interface is the area below the viewport in Abaqus/CAE. Chances are the message area is currently displayed here. If you click the box with '>>>' on it you will be able to type in commands. We will use this to test a few different Python commands in the next chapter. For now I wish to make you aware that it is possible to run a script from here using the execfile() command.

Type in

```
Execfile('cantilever_beam.py')
```

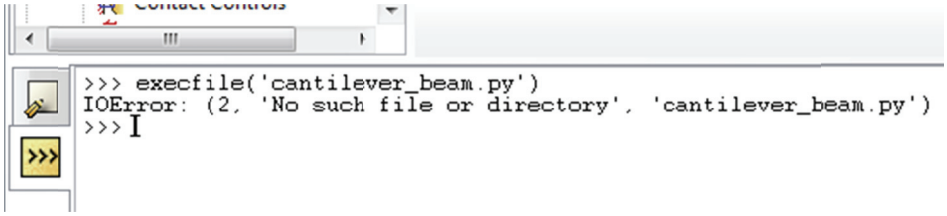
The file you've passed as an argument to execfile() needs to be present in the current work directory for Abaqus, otherwise you need to spell out the full path such as:

```
Execfile('C:\Users\Gautam\Desktop\Book\cantilever_beam.py')
```

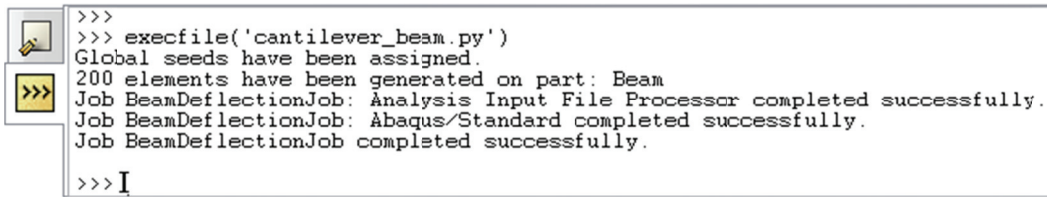
By default the work directory is C:\Temp although you can change it using **File > Set Work Directory..**

If the file is not in the current work directory and you did not specify the full path, Abaqus will not find the script and will display an IOError.

```
IOError: (2, 'No such file or directory', 'cantilever_beam.py')
```



If the file is present in the work directory, or you specify the full path, the script executes successfully.



2.6 Conclusion

This chapter has presented to you some of the various ways in which scripts can be run. You may choose the appropriate method based on the task at hand, or feel free to go with personal preference.

3

Python 101

3.1 Introduction

In the cantilever beam example of Chapter 1, we began by creating the entire model in Abaqus/CAE. We then opened up a new file and ran a script which accomplished the exact same task. How exactly did the script work and what did all those code statements mean? Before we can start to analyze this, it is necessary to learn some basic Python syntax. If you have any programming experience at all, this chapter should be a breeze.

3.2 Statements

Python is written in the form of code statements as are other languages. However you do not need to put a semi-colon at the end of each statement. What the Python interpreter looks for are carriage returns (that's when you press the ENTER key on the keyboard). As long as you hit ENTER after each statement so that the next one is on a new line, the Python interpreter can tell where one statement ends and the other begins.

In addition statements within a code block need to be indented, such as statements inside a FOR loop. In languages such as C++ you use curly braces to signal the beginning and end of blocks of code whereas in Python you indent the code. Python is very serious about this, if you don't indent code which is nested inside of something else (such as statements in a function definition or a loop) you will receive a lot of error messages.

Within a statement you can decide how much whitespace you wish to leave. So a=b+c can be written as a = b + c (notice the spaces between each character)

3.3 Variables and assignment statements

In some programming languages such as C++ and Java, variables are strongly typed. This means that you don't just name a variable; you also declare a type for the variable. So for

example if you were to create an integer variable ‘x’ in C++ and assign it a value of 5, your code would look something like the following:

```
int x;
x=5;
```

However Python is not strongly typed. This means you don’t state what type of data the variable holds, you simply give it a name. It could be an integer, a float or a String, but you wouldn’t tell Python, it would figure it out on its own. So if you were to create an integer variable x in Python and assign it a value of 5 you would simply write:

```
x=5
```

In addition Python doesn’t mind if you try to do things like multiplying a whole number with a float. Some languages object to this type of mixing and require an explicit cast. Python is also able to recognize String variables, and concatenates them if you add them. So a statement like

```
greeting = ‘h’ + ‘ello’
```

stores the value ‘hello’ in the variable ‘greeting’.

Let’s work through an example to understand some of these concepts.

Example 4.1 - Variables

Open up Abaqus CAE. In the lower half of the window below the viewport you see the message area. If you look to the left of the message area you see two tabs, one for “Message area” and the other for “Kernal Command Line Interface”.

Click the second one. You see the kernel command prompt which is a “>>>” symbol.

Type the following lines, hitting the ENTER key on your keyboard after each.

```
>>> length = 10
>>> width = 4
>>> area = length * width
>>> print area
```

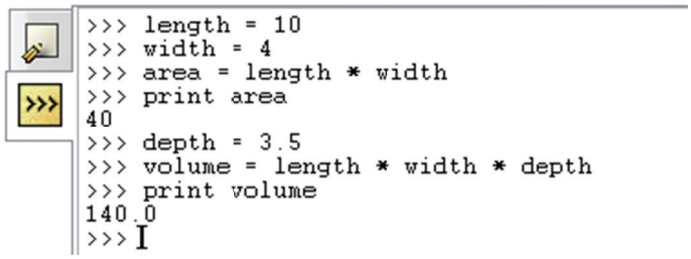
The number 40 is displayed. Since we set length to 10 and width to 4, the area being the product of the two is 40. The print statement displays the value stored in the area variable. The following image displays what you should see on your own screen.

So you see the Python interpreter realized that the variables 'length' and 'width' store integers without you needing to specify what type of variables they are. In addition when assigning their product to the variable 'area', it decided for itself that 'area' was also an integer.

What if you had combined integers and floats? Add on the following statements:

```
>>> depth = 3.5
>>> volume = length * width * height
>>> print volume
```

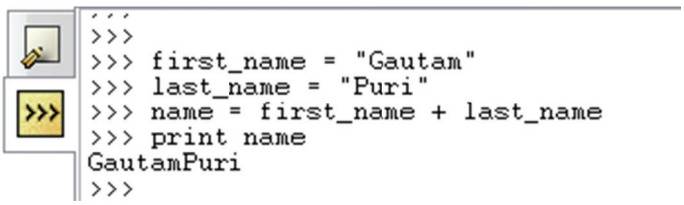
The output is 140.0 . Note the ".0" at the end. Since your height variable was a float (number with decimal point in layman terms), the volume variable is also a float, even though two of its factors 'length' and 'width' are integers.



Let's experiment with Strings. Add the following lines

```
>>> first_name = "Gautam"
>>> last_name = "Puri"
>>> name = first_name + last_name
```

The output is "GautamPuri". Notice that we did not tell Python that 'first_name' and 'last_name' are String variables, it figured it out on its own. Also when we added them together, Python concatenated them together.



As you can tell from this example, not having to define variable types makes it a lot less painful to type code in Python than in a language such as C++. This also saves a lot of heartache when dealing with instances of classes so that you don't have to define each variable as being an object of a class. If you don't know what classes, instances and objects are, you will find out in the section on "Classes" a few pages down the line. But first let's talk about lists and dictionaries.

3.4 Lists

Arrays are a common collection data type in just about every high level programming language so I expect you've dealt with them before and know why they're useful. You aren't required to use them to write Abaqus scripts, but chances are you will want to store information in similar collections in your scripts. Let's explore a collection type in Python known as a List.

In a list you store multiple elements or data values and can refer to them with the name of the list variable followed by an index in square brackets []. The lowest index is 0. Note that you can store all kinds of data types, such as integers, floats, Strings, all in the same list. This is different from languages such as C, C++ and Java where all array members must be of the same data type. Lists have many built-in functions, some of which are:

- len() – returns the number of elements in the list
- append(x) – adds x to the end of the list making it the last element

- remove(y) – removes the first occurrence of y in the list
- pop(i) – removes the element at index [i] in the list, also returns it as the return value

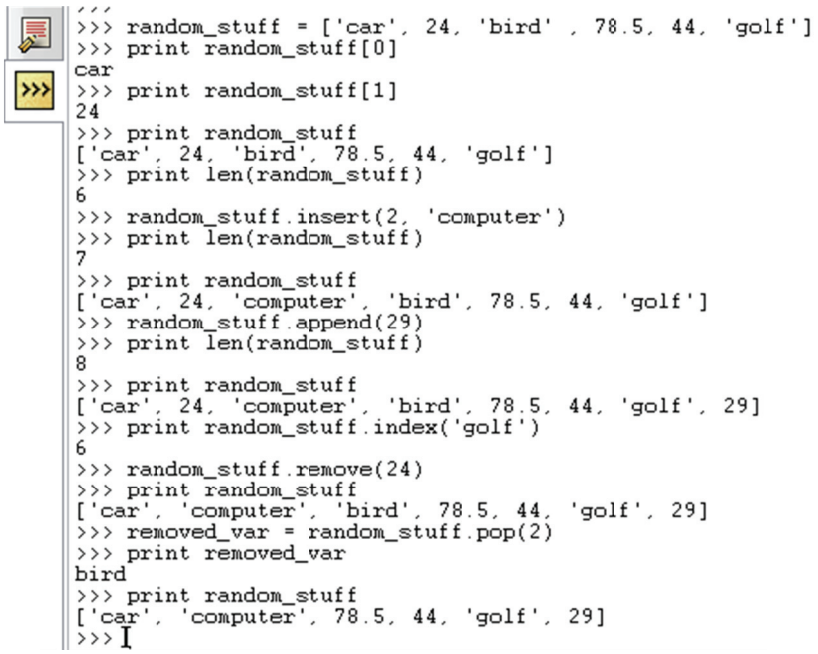
Let's work through an example.

Example 4.2 - Lists

In the 'Kernel Command Line Interface' tab of the lower panel of the window, type in the following statements hitting ENTER after each.

```
>>>random_stuff = ['car', 24, 'bird' , 78.5, 44, 'golf']
>>> print random_stuff[0]
>>> print random_stuff[1]
>>> print random_stuff
>>> print len(random_stuff)

>>> random_stuff.insert(2, 'computer')
>>> print len(random_stuff)
>>> print random_stuff
>>> random_stuff.append(29)
>>> print len(random_stuff)
>>> print random_stuff
>>> random_stuff.remove(24)
>>> print random_stuff

>>> removed_var = random_stuff.pop(2)
>>> print removed_var
>>> print random_stuff
```

Your output will be as displayed the following figure. Note that the lowest index is 0, not 1, which is why random_stuff[0] refers to the first element 'car'. The len() function returns the number of elements in the list. The append() function adds on whatever is passed to it as an argument to the end of the list. The remove() function removes the element that matches the argument you pass it. And the pop() function removes the element at the index position you pass it as an argument.



```
>>> random_stuff = ['car', 24, 'bird' , 78.5, 44, 'golf']
>>> print random_stuff[0]
car
>>> print random_stuff[1]
24
>>> print random_stuff
['car', 24, 'bird', 78.5, 44, 'golf']
>>> print len(random_stuff)
6
>>> random_stuff.insert(2, 'computer')
>>> print len(random_stuff)
7
>>> print random_stuff
['car', 24, 'computer', 'bird', 78.5, 44, 'golf']
>>> random_stuff.append(29)
>>> print len(random_stuff)
8
>>> print random_stuff
['car', 24, 'computer', 'bird', 78.5, 44, 'golf', 29]
>>> print random_stuff.index('golf')
6
>>> random_stuff.remove(24)
>>> print random_stuff
['car', 'computer', 'bird', 78.5, 44, 'golf', 29]
>>> removed_var = random_stuff.pop(2)
>>> print removed_var
bird
>>> print random_stuff
['car', 'computer', 78.5, 44, 'golf', 29]
>>>
```

3.5 Dictionaries

Dictionaries are a collection type, just as lists are, but with a slightly different feel and syntax. You do not really need to create your own dictionaries in order to write scripts in Abaqus, you can accomplish most tasks with a list, but you never know when you might prefer to use a dictionary. More importantly though, Abaqus stores a number of its own constructs in the form of dictionaries, and you will be accessing these regularly, hence knowing what dictionaries are will give you a better understanding of scripting.

Dictionaries are sets of key:value pairs. To access a value, you use the key for that value. This is analogous to using an index position to access the data within a list. The difference is that keeping track of the key to access a value may be easier in a certain situation than remembering the index location of a value in a list. Since there are no index positions, dictionaries are unordered.

To remove a key:value pair, you use the del command. To remove all the key:value pairs, you use the clear command.

Aside: If you've worked with the programming language PERL, dictionaries are very similar to the hash collections. If you're coming from a Java environment, dictionaries are similar to the Hashtable class.

An example should make things clear.

Example 4.3 – Dictionaries

In the 'Kernel Command Line Interface', type in the following statements hitting ENTER after each. You will see an output after each print statement.

```
>>>names_and_ages = {'John':23, 'Rahul':15, 'Lisa':55}
>>> print names_and_ages['John']
>>> print names_and_ages['Rahul']
>>> print names_and_ages
>>> del names_and_ages['John']
>>> print names_and_ages
>>> names_and_ages.clear()
>>> print names_and_ages
```

Here names_and_ages is your dictionary variable. In it you store 3 keys, 'John', 'Rahul' and 'Lisa'. You store their ages as the values. This way if you wish to access Lisa's age, you would write names_and_ages['Lisa'].

The del command removes the key:value pair 'John':23, leaving only Rahul and Lisa. The clear command removes all the key value pairs leaving you with an empty dictionary {}.

Note that since the dictionary is unordered, the first statement could instead have been written as

```
>>> names_and_ages = {'Rahul':15, 'Lisa':55, 'John':23}
```

and it would have made no difference since your values (ages) are still bound to the correct keys (names).

The following image displays what you should see.

```
>>>
>>> names_and_ages = {'John':23, 'Rahul':15, 'Lisa':55}
>>> print names_and_ages['John']
23
>>> print names_and_ages['Rahul']
15
>>> print names_and_ages
{'Lisa': 55, 'John': 23, 'Rahul': 15}
>>> del names_and_ages['John']
>>> print names_and_ages
{'Lisa': 55, 'Rahul': 15}
>>> names_and_ages.clear()
>>> print names_and_ages
{}
>>>
```

So how does Abaqus use dictionaries?

You're probably wondering when you would actually use dictionaries. You will be using them all the time, and already did so more than once in the cantilever beam example of Chapter 1 (Example 1.2), except you didn't know it at the time. Here's a block of code from the example.

```
# ------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# ------------------------------------------------------
```

Look closely at the statement

```
beamModel = mdb.models['Cantilever Beam']
```

Here you see the model database 'mdb' has a property called 'models'. This property is a dictionary object containing a key:value pair for each model you create. The model name itself is the 'key', and the value is an instance of the model object.

You know that the syntax to access a value in a dictionary is *dictionary_name['key']*. So when you want the script to refer to the cantilever beam model you say *models['Cantilever Beam']*.

To be a little more precise, models in not exactly a dictionary object but a subclass of a dictionary object. What does that mean? Well, to put it simply, it means that the programmers at Abaqus created a new class that has the same properties and methods as

dictionary, but also has one or more new properties and methods that they defined. For example the changeKey() method that changes the name of the key from 'Model-1' to 'Cantilever Beam' is not native to Python dictionaries, it has been created by programmers at Abaqus. You don't have to worry about how they did it unless you are a computer science buff, in which case google 'subclassing in Python'. As far as a user/scripter is concerned, the 'models' object works similar to a dictionary object with a few enhancements. Also in Abaqus these enhanced dictionaries are referred to as 'repositories'. You will hear me use this word a lot when we start dissecting scripts.

Let's look at another block of code from Example 1.2.

```
# -------------------------------------------------------
# Create the history output request

# we try a slightly different method from that used in field output request
# create a new history output request called 'Default History Outputs' and assign
both the step and the variables
beamModel.HistoryOutputRequest(name='Default History Outputs', createStepName='Apply
Load', variables=PRESELECT)

# now delete the original history output request 'H-Output-1'
del beamModel.historyOutputRequests['H-Output-1']

# -------------------------------------------------------
```

Look closely at the statement

```
del beamModel.historyOutputRequests['H-Output-1']
```

Notice that your model beamModel has a dictionary or 'repository' (subclass of a dictionary) called historyOutputRequests. One of the key:value pairs has a key 'H-Output-1', and is referred to as *historyOutputRequests['H-Output-1']*. In the Abaqus Scripting Interface you will often find aspects of your model stored in repositories. For the record, in this statement the 'H-Output-1' key:value pair in the repository is being deleted using the del command.

3.6 Tuples

(Section removed from Preview)

3.7 Classes, Objects and Instances

When you run scripts in Abaqus you invariably use built-in methods provided by Abaqus to perform certain tasks. All of these built-in methods are stored in containers called classes. You often create an "instance" of a class and then access the built-in methods which belong to the class or assign properties using it. So it's important for you to have an understanding of how this all works.

Python is an object oriented language. If you've programmed in C++ or Java you know what object oriented programming (OOP) is all about and can breeze through this section. On the other hand if you're used to procedural languages such as C or MATLAB you've probably never worked with objects before and the concept will be a little hard to grasp at first. (Actually MATLAB v2008 and above supports OOP but it's not a feature known by the majority of its users).

For the uninitiated, a class is a sort of container. You define properties (variables) and methods (functions) for this class, and the class itself becomes a sort of data type, just like integer and String are data types. When you create a variable whose data type is the class you've defined, you end up creating what is called an object or an instance of the class. The best way to understand this is through an example.

Example 4.5 – 'Person' class

In the following example, assume we have a class called 'Person'. This class has some properties, such as 'weight', 'height', 'hair' color and so on. This class also has some methods such as 'exercise()' and 'dyeHair()' which cause the person to lose weight or change hair color.

Once we have this basic framework of properties and methods (called the class definition), we can assign an actual person to this class. We can say Gary is a 'Person'. This means Gary has properties such as height, weight and hair color. We can set Gary's height by using a statement such as Gary.height = 68 inches. We can also make Gary exercise by saying Gary.exercise() which would cause Gary.weight to reduce. Gary is "an object of type Person" or "an instance of the Person class".

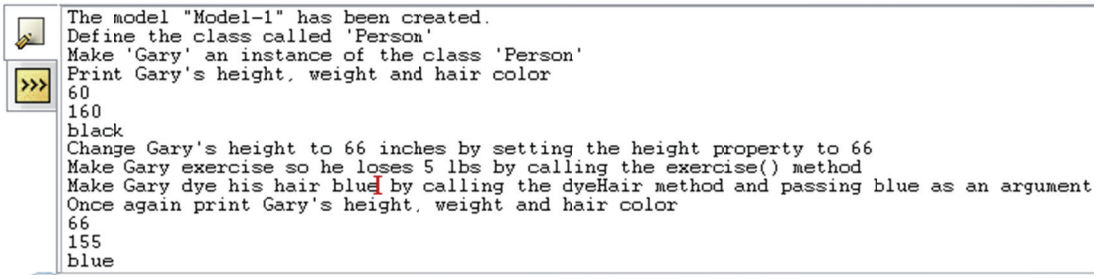
Open up notepad and type out the following script

```
print "Define the class called 'Person'"
```

```
class Person:
        height = 60
        weight = 160
        hair_color = "black"

        def exercise(self):
                self.weight = self.weight - 5

        def dyeHair(self, new_hair_color):
                self.hair_color = new_hair_color

print "Make 'Gary' an instance of the class 'Person'"
Gary = Person()

print "Print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color

print "Change Gary's height to 66 inches by setting the height property to 66"
Gary.height = 66

print "Make Gary exercise so he loses 5 lbs by calling the exercise() method"
Gary.exercise()

print "Make Gary dye his hair blue by calling the dyeHair method and passing blue as
an argument”
Gary.dyeHair('blue')

print "Once again print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color
```

Open a new file in Abaqus CAE (**File > New Model Database > With Standard/Explicit Model**). In the lower half of the window, make sure you are in the “Message Area” tab, not the “Kernel Command Line Interface” tab. The print statements in our script will display here in the “message area” that’s why you want it to be visible.

Run the script you just typed out (**File > Run Script…**). Your output will be as displayed in the following figure.



```
The model "Model-1" has been created.
Define the class called 'Person'
Make 'Gary' an instance of the class 'Person'
Print Gary's height, weight and hair color
60
160
black
Change Gary's height to 66 inches by setting the height property to 66
Make Gary exercise so he loses 5 lbs by calling the exercise() method
Make Gary dye his hair blue by calling the dyeHair method and passing blue as an argument
Once again print Gary's height, weight and hair color
66
155
blue
```

Let's analyze the script in detail. The first statement is

```
print "Define the class called 'Person'"
```

This basically prints "Define the class called 'Person'" in the message window using the 'print' command. Hence that is the first message displayed. The following statements define the class:

```
class Person:
        height = 60
        weight = 160
        hair_color = "black"

        def exercise(self):
                self.weight = self.weight - 5

        def dyeHair(self, new_hair_color):
                self.hair_color = new_hair_color
```

A class named 'Person' has been created. It has been given the properties (variables) 'height', 'width' and 'hair_color', which have been assigned initial values of 60 inches, 160 lbs, and the color black.

In addition two methods (functions) have been defined, 'exercise()' and 'dyeHair()'. The 'exercise()' method causes the weight of the person to decrease by 5 lbs. The 'dyeHair()' function causes 'hair_color' to change to whatever color is passed to that function as the argument 'new_hair_color'.

What's with the word 'self'? In Python, every method in a class receives 'self' as the first argument, that's a rule. The word 'self' refers to the instance of the class which will be created later. In our case this will be 'Gary'. When we create 'Gary' as an instance of the 'Person' class, *self.weight* translates to *Gary.weight* and *self.hair_color* translates to *Gary.hair_color*. In object oriented languages like C++ and Java you do not pass self as

an argument, this is a feature unique to the Pythons syntax and might even be a little annoying at first.

```
print "Make 'Gary' an instance of the class 'Person'"
Gary = Person()
```

These statements define Gary as an instance of the Person class, and also print a comment to the message area indicating this fact.

```
print "Print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color
```

We then display Gary's height, weight and hair_color which are currently default values. Notice how we refer to each variable with the instance name followed by a dot "." symbol followed by the variable name. The format is *InstanceName.PropertyName*.

These statements make the following lines appear on the screen:

```
Print Gary's height, weight and hair color"
60
160
black
```

```
print "Change Gary's height to 66 inches by setting the height property to 66"
Gary.height = 66
```

We now change Gary's height to 66 inches by using an assignment statement on the 'Gary.height' property. We print a comment regarding this to the message area.

```
print "Make Gary exercise so he loses 5 lbs by calling the exercise() method"
Gary.exercise()
```

These lines call the exercise function and display a comment in the message area. Notice that you use the format *InstanceName.MethodName()*. Although we don't appear to pass any arguments to the function (there's nothing in the parenthesis), internally the Python interpreter is passing the instance 'Gary' as an argument. This is why in the function definition we had the word 'self' listed as an argument. Why does the interpreter pass 'Gary' as an argument? Because you could potentially define a number of instances of the Person class in addition to Gary, such as 'Tom', 'Jill', 'Mr. T', and they will all have

the same ‘exercise()’ method. So then if you were to call *Tom.exercise()*, it would be Tom’s weight that would reduce while Gary’s would remain unaffected.

If you look once again at the definition of the ‘exercise()’ method in the Person class, you’ll notice that it decreases the weight of the instance by 5 lbs. So Gary’s weight should now be 155 lbs, down 5 lbs from before.

```
print "Make Gary dye his hair blue by calling the dyeHair method and passing blue as
an argument”
Gary.dyeHair('blue')
```

These lines call the ‘dyeHair()’ function and display a comment in the message area. The difference you notice between the ‘exercise()’ and ‘dyeHair()’ functions is that you pass a hard argument to ‘dieHair()’ telling it exactly what color you wish to dye the individuals hair. Internally an argument of ‘self’ is also passed.

Take another look at the definition of the ‘dyeHair()’ method in the ‘Person’ class. You’ll notice that the variable being passed as an argument is assigned to the ‘hair_color’. So Gary’s hair color should now have changed from black to blue.

```
print "Once again print Gary's height, weight and hair color"
print Gary.height
print Gary.weight
print Gary.hair_color
```

We print out Gary’s height, weight and hair color again to notice the changes. The ‘Gary.height’ statement was used to reset his height to 66 inches, the ‘exercise()’ method was used to reduce his weight to 155 lbs, and the ‘dyeHair(‘blue’)’ method should have changed his hair color to blue. These print statements display the property values in the message area. The output is what you expect:

```
Once again print Gary’s height, weight and hair color
66
155
blue
```

Hopefully this example has made the concept of classes and instances clear to you. There’s a lot more to OOP than this, we’ve only touched the surface, but that’s because you only need a basic understanding of OOP to write Abaqus scripts. In none of our examples will you actually define a new class of your own.

So why learn about classes, objects and instances?

(Removed from Preview)

Abstraction in OOP

(Removed from Preview)

3.8 What's next?

In this chapter you learned :

- how to define variables and write code statements,
- how to create collection types – lists, dictionaries, and tuples,
- object oriented programming (OOP) concepts – classes, instances, data abstraction

You also referred to code snippets from the cantilever beam example from Chapter 1 to see the syntax in action.

You now understand some of the Python syntax behind much of Example 1.2. However you still don't understand the Abaqus specific commands and methods that were used. In the next chapter we're going to take a closer look at the cantilever beam example and try to make sense of it all.

4

The Basics of Scripting – Cantilever Beam Example

4.1 Introduction

Now that you have the required understanding of Python syntax, we can plunge into scripting. Every script you write will perform a different task and no two scripts will be alike. However they all follow the same basic methodology. The best way to understand this is to go through the cantilever beam script in detail.

4.2 A basic script

Since we already have the cantilever beam example from Chapter 2 we shall work our way through it, statement by statement. Not only will you understand exactly what is going on in the script, you will also learn some of the most important methods that you will likely use in every script you write.

Example 4.1 – Cantilever Beam

For your convenience a copy of the code from Chapter 2 has been listed here.

```
# ******************************************************************************
# Cantilever Beam bending under the action of a uniform pressure load

# ******************************************************************************

from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ------------------------------------------------------------------------
```

```
# Create the model
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']

# ------------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))

# b) Create a 3D deformable part named "Beam" by extruding the sketch
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)

# ------------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs
# modulus and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ),       ))
beamMaterial.Elastic(table=((200E9, 0.29), ))

# ------------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the beam
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')

# Assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')

# ------------------------------------------------------
# Create the assembly


                        (Statements removed from preview)


# ------------------------------------------------------------------------
```

```
# Create the step

                          (Statements removed from preview)

# -------------------------------------------------------------------------
# Create the field output request


                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Create the history output request


                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Apply pressure load to top surface


                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Apply encastre (fixed) boundary condition to one end to make it cantilever


                          (Statements removed from preview)


# -------------------------------------------------------------------------
# Create the mesh


                          (Statements removed from preview)


# -------------------------------------------------------------------------
```

```
# Create and run the job



                        (Statements removed from preview)


# ------------------------------------------------------------------------
# Post processing

import visualization

beam_viewport = session.Viewport(name='Beam Results Viewport')
beam_Odb_Path = 'CantileverBeamJob.odb'
an_odb_object = session.openOdb(name=beam_Odb_Path)
beam_viewport.setValues(displayedObject=an_odb_object)
```

4.3 Breaking down the script

The script executes from top to bottom in Python. I have included comments all over the script to explain what's going on. Lines that start with the hash (#) symbol are treated as comments by the interpreter. Make it a point to comment your code so you know what it means when you look at it after a few months or another member of your team has to continue what you started.

Observe the layout of the script. I have divided it into blocks or chunks of code clearly demarcated by:

```
#-------------------------------------------------------
# comment describing the block of code
```

Try reading these comments. You will realize that the script follows these steps:

1. Initialization (import required modules)
2. Create the model
3. Create the part
4. Define the materials
5. Create solid sections and make section assignments
6. Create an assembly
7. Create steps
8. Create and define field output requests
9. Create and define history output requests
10. Apply loads

11. Apply boundary conditions
12. Meshing
13. Create and run the job
14. Post processing

Let's explore each code chunk one at a time.

4.3.1 Initialization (import required modules)

The code block dealing with this step is listed below:

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

We begin the script using a couple of 'from-import' statement.

The first import statement:

```
from abaqus import *
```

imports the **abaqus** module and creates references to all public objects defined by that module. Thus it makes the basic Abaqus objects accessible to the script. One of the things it provides access to is a default model database which is referred to by the variable **mdb**. You use this variable **mdb** in the next block of code which is the 'create the model' block. You need to insert this import statement in every Abaqus script you write.

The second import statement:

```
from abaqusConstants import *
```

is for making the symbolic constants defined by the Abaqus Scripting Interface available to the script. What are symbolic constants? They are variables with a constant value (hence the term constant) that have been given a name that makes more sense to a user (hence the term symbolic) but have some meaning to Abaqus. Internally they might be integer or float variables. But for the sake of clarity of code they are displayed as a word in the English language. Since they are constants they cannot be modified

We use symbolic constants in the script. Look at the relevant lines in the script where the part is created. Notice the statement:

```
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
```

Both **THREE_D** and **DEFORMABLE_BODY** are symbolic constants defined in the **abaqusConstants** module. So if we did not import this module into our script we would get an error as the interpreter would not recognize these symbolic constants. So place this import statement in every script you write.

The third import statement:

```
import regionToolset
```

imports the **regionToolset** module so you can access its methods through the script. If you look at the 'create the loads' block, you will notice the statement:

```
top_face_region=regionToolset.Region(side1Faces=top_Plate)
```

We are using the **Region()** method defined in the **regionToolset** module. Hence the module needs to be imported otherwise you will receive an error. I tend to place this import statement in every script I write, whether or not the **Region()** method is used, just to be on the safe side.

Basically every script should have these 3 import statements placed in it at the top. You may not always need them, but by including them you spend less time thinking about whether or not you need them and more time writing useful code.

The fourth statement:

```
session.viewports['Viewport:1'].setValues(displayedObject=None)
```

blanks out the viewport. The viewport is the window in the Abaqus/CAE that displays the part you are working on. It allows Abaqus to display information to you visually. The viewport object is the object where the information about the viewport is stored such as what to display and how to do so.

The default name for the viewport is 'Viewport:1'. This is not only the name displayed to the user, it is the key for that viewport in the **viewports** dictionary/repository. Hence we refer to the viewport with the **viewports['Viewport:1']** notation. The method

setValues() is a method of the **viewport** object that can be used to modify the viewport. It accepts two parameters, the **displayedObject** which defines what is displayed, and the **displayMode** which defines the layers (more about that later). When we set the **displayedObject** to **None**, that causes an empty viewport to be displayed.

4.3.2 Create the model

The following block creates the model

```
# ------------------------------------------------------------------------
# Create the model
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
beamModel = mdb.models['Cantilever Beam']
```

As stated before, the variable **mdb** provides access to a default model database. This variable is made available to the script thanks to the

```
from abaqus import *
```

import statement we used earlier, hence you don't define it yourself.

The default model in Abaqus is always named 'Model-1', which is why when you open a new file you always see 'Model-1' in the model database tree on the left in the GUI.

The first statement:

```
mdb.models.changeKey(fromName='Model-1', toName='Cantilever Beam')
```

changes the name of the model from the default of 'Model-1' to 'Cantilever Beam'. **changeKey()** is a method of models which is in the model database, hence we refer to it using **mdb.models.changeKey()**.

If you recall from Chapter 3, the **models** repository is a subclass of a dictionary object which keeps track of model objects. As explained before, a subclass means that it has the same properties and methods of the dictionary object along with a few more properties and methods, such as **changeKey()**, that developers at SIMULIA decided to add in. The model name 'Model-1' is the key, while the value is a **model** object. The **changeKey()** method which is not native to Python essentially allows us to change the key to 'Cantilever Beam' while referring to the same model object.

The second statement:

```
beamModel = mdb.models['Cantilever Beam']
```

assigns our model to the **beamModel** variable. This is so that in future we do not have to keep referring to it as **mdb.models[‘Cantilever Beam’]** but can instead just call it **beamModel**. Look at the ‘create the part’ block and notice the statement

```
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
```

Don’t worry about what it means just yet, I only want to point out that if we did not define the variable **beamModel**, then the same statement would have to be written as:

```
beamProfileSketch = mdb.models[‘Cantilever Beam’].
                        ConstrainedSketch (name=’Beam CS Profile, sheetSize=5)
```

which is a little bit longer. This type of syntax will get longer as we refer to properties and objects nested further down.

Of course you could choose to write things the long way, or you could do it my way.

4.3.3 Create the part

The following block of code creates the part

```
# ------------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the beam cross section using rectangle tool
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                sheetSize=5)
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))

# b) Create a 3D deformable part named "Beam" by extruding the sketch
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)
```

The first two statements

```
import sketch
import part
```

import the **sketch** and **part** modules into the script, thus providing access to the objects related to sketches and parts. As such you shouldn't be able to create a sketch or a part without these import statements but honestly if you leave them out in most cases Abaqus figures out what you are trying to do and appears to import these modules automatically without complaining. It is however recommended that you stay in the habit of including them because it's good programming practice and because you never know if an older or newer version of Abaqus will throw an error.

The statement

```
beamProfileSketch = beamModel.ConstrainedSketch(name='Beam CS Profile',
                                                 sheetSize=5)
```

creates a constrained sketch object by calling the ConstrainedSketch() method of the Model object. The sketch module defines ConstrainedSketch objects. The first argument is the **name** you wish to give the sketch, we're calling it 'Beam CS Profile'. This is used as the repository key given to our **ConstrainedSketch** object, just as 'Cantilever Beam' is the key for our model object. The second argument is the default **sheetsize**, which is a property you defined when manually performing the cantilever beam simulation in Abaqus/CAE. It sets the approximate size of the sheet, and therefore the grid you see when you are in the sketcher. Of course when you're working in a script the sheetsize isn't really important, that only helps you see things better when working in the GUI, but it's a required paramenter to the **ConstrainedSketch()** method hence you must give it a value. Note that the statement can be written without the words 'name' and 'sheetSize' as:

```
beamProfileSketch = beamModel.ConstrainedSketch('Beam CS Profile', 5)
```

It means the same thing to the interpreter; it just isn't as clear to someone reading your script. Also you'll have to make sure the arguments are passed in the correct order as is required by the method as stated in the documentation.

The statement

```
beamProfileSketch.rectangle(point1=(0.1,0.1), point2=(0.3,-0.1))
```

uses the **rectangle()** method of the **ConstrainedSketch** object to draw a rectangle on the sketch plane. The two parameters are the coordinates of the top left and bottom right

corners of the rectangle. Note that the statement can also be written without the words **point1** and **point2** as:

```
beamProfileSketch.rectangle((0.1,0.1), (0.3,-0.1))
```

The statement

```
beamPart=beamModel.Part(name='Beam', dimensionality=THREE_D,
                                     type=DEFORMABLE_BODY)
```

uses the **Part()** method to create a **Part** object and place it in the parts repository. The first parameter 'Beam' is its **name** and its key in the repository. The second parameter, **dimensionality**, is set to a symbolic constant **THREE_D** which defines it to be a 3D part. It is defined to be of the **type** deformable body using the **DEFORMABLE_BODY** symbolic constant. In subsequent chapters you will define different parameters in place of these depending on the simulation. The created part instance is stored in the **beamPart** variable. If you haven't already guessed, the statement can also be written without the words **name**, **dimensionality**, and **type** as

```
beamPart=beamModel.Part('Beam', THREE_D, DEFORMABLE_BODY)
```

The statement

```
beamPart.BaseSolidExtrude(sketch=beamProfileSketch, depth=5)
```

creates a **Feature** object by calling the **BaseSolidExtrude()** method. What is a **Feature** object? Well, Abaqus is a feature based modeling system. The **Feature** object contains the parameters specified by the user, as well as the modifications made to the model by Abaqus based on those parameters. The **Feature** object is defined in the **Part** module hence you do not use an 'import feature' statement. The **BaseSolidExtrude()** method is used to create extrusions. The first parameter passed to it is our **ConstrainedSketch** object **beamProfileSketch**. Note that this must be a closed profile. The second parameter is the **depth** to which we wish to extrude our profile sketch. The statement can be written without the keywords **sketch** and **depth** as:

```
beamPart.BaseSolidExtrude(beamProfileSketch, 5)
```

4.3.4 Define the materials

The following block creates the material

```
# ------------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs
# modulus and poissons ratio
beamMaterial = beamModel.Material(name='AISI 1005 Steel')
beamMaterial.Density(table=((7872, ), ))
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

`import material`

This statement imports the **material** module into the script providing access to objects and methods related to materials.

`beamMaterial = beamModel.Material(name='AISI 1005 Steel')`

This statement creates a **Material** object using the **Material()** method and places it in the **materials** repository. The parameter passed to the **Material()** method is the name given to the material, and the key used to refer to it in the **materials** repository. The **Material** object is assigned to the variable **beamMaterial**.

`beamMaterial.Density(table=((7872, ), ))`

This statement creates a **Density** object which specifies the density of the material by using the **Density()** method. The **Density** object is defined in the **material** module, hence you do not use an 'import density' statement. The argument passed to the **Density** method is supposed to be a table. Why a table? Well you might have a density that depends on temperature. In which case you would have a table in the form *((density1, temperature1), (density2,temperature2), (density3,temperature3))* and so on…

In our case we have one density which is not temperature dependent, but we must use the same format. So we can't say *table=7872*, we need to write *table=((7872,),)* where we leave a space after the first comma for *temperature1* (or rather the lack of it), and a space after the second comma for *(denstiy2, temperature2)*.This probably looks a little strange, and you will often generate a lot of syntax errors typing the wrong number of commas or parenthesis, so be aware of that. For the record, we can leave out the word 'table', but all the parentheses and commas in the statement will remain as they are:

```
beamMaterial.Density(((7872, ), ))
```

The statement:

```
beamMaterial.Elastic(table=((200E9, 0.29), ))
```

creates an **Elastic** object which specifies the elasticity of the material by using the **Elastic()** method. The **Elastic** object is defined in the **material** module, hence you do not use an *import elastic* statement. The argument passed to the **Elastic()** method must be a table just like the argument to the **Density()** method. The table must be of the form *((YM1, PR1), (YM2, PR2), (YM3, PR3))* and so on where *YM* is Young's modulus and *PR* is Poisson's ratio. For our material we have only one Young's modulus and one Poisson's ratio so we write *table=((200E9, 0.29),)* leaving a second comma there to indicate the spot for (YM2, PR2). The statement can be written without the keyword 'table' as:

```
beamMaterial.Elastic(((200E9, 0.29), ))
```

4.3.5 Create solid sections and make section assignments

The following code block creates the sections and makes assignments

```
# ------------------------------------------------------------------------
# Create solid section and assign the beam to it

import section

# Create a section to assign to the beam
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')

# Assign the beam to this section
beam_region = (beamPart.cells,)
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')
```

```
import section
```

This statement imports the **section** module making its properties and methods accessible to the script.

```
beamSection = beamModel.HomogeneousSolidSection(name='Beam Section',
                                                material='AISI 1005 Steel')
```

This statement creates a **HomogeneousSolidSection** object using the **HomogeneousSolidSection()** method. These are defined in the section module. The first parameter given to the method is **name**, which is used as the repository key. The second parameter is **material**, which has been defined in the 'define the materials' code block.

Note that this **material** parameter must be a String, it cannot be a **Material** object. That means we cannot say *material=beamMaterial* even though we had defined the **beamMaterial** variable to point to our beam material, because **beamMaterial** is a **Material** object. 'AISI1005 Steel' on the other hand is a String, and it is the key assigned to that material in the **materials** repository.

The statement

```
beam_region = (beamPart.cells,)
```

is used to find the **cells** of the beam. The **cell** object defines the volumetric regions of a geometry. **Part** objects have cells. **beamPart.cells** refers to the Cell object that contains the information about the cells of the beam.

Notice however that there is a comma after beamPart.cells. This is because we are trying to create a variable which is a **Region** object. A **Region** object is a type of object on which you can apply an attribute. You can use a **Region** object to define the geometry for a section assignment, or a load, or a boundary condition, or a mesh, basically it forms a link between the geometry and the applied attribute. A **Region** object can be a sequence of **Cell** objects. In fact it can be a sequence of quite a few other objects, including **Node** objects, **Vertex** objects, **Edge** objects and **Face** objects. In our script we are assigning a **Cell** object to it. But since it needs to be a sequence of **Cell** objects, not just one **Cell** object that we are providing, we stick the comma at the end to make it a sequence. We then assign it to the variable **beam_region**.

Why exactly are we creating a **Region** object? Because we need it for the next statement where we use the **SectionAssignment()** method.

```
beamPart.SectionAssignment(region=beam_region, sectionName='Beam Section')
```

This statement creates a **SectionAssignment** object, which is an object that is used to assign sections to a part, an assembly or an instance. This is done using the **SectionAssignment()** method. Its first parameter is a **region**, in this case the region is the entire part. We have already created a region in the previous statement called **beam_region** using all the cells of the part, and we now this region as our first parameter. The second argument is the **name** we wish to give the section, which is also the key it will be assigned in the **sections** repository. This argument must be a String, therefore we

cannot use the variable **beamSection** which is a **Section** object, but rather its name/key. The statement can be written without the keywords **region** and **sectionName** as:

```
beamPart.SectionAssignment(beam_region, 'Beam Section')
```

4.3.6 Create an assembly

(Section removed from Preview)

4.3.7 Create steps

(Section removed from Preview)

4.3.8 Create and define field output requests

(Section removed from Preview)

4.3.9 Create and define history output requests

(Section removed from Preview)

4.3.10 Apply loads

(Section removed from Preview)

4.3.11 Apply constraints/boundary conditions

(Section removed from Preview)

4.3.12 Mesh

(Section removed from Preview)

4.3.13 Create and run the job

(Section removed from Preview)

4.3.14 Post processing

The following code performs some post processing tasks:

```
# ------------------------------------------------------------------------
# Post processing

import visualization

beam_viewport = session.Viewport(name='Beam Results Viewport')
beam_Odb_Path = 'CantileverBeamJob.odb'
an_odb_object = session.openOdb(name=beam_Odb_Path)
beam_viewport.setValues(displayedObject=an_odb_object)
beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

```
import visualization
```

This statement imports the **visualization** module. This allows the script to access methods that replicate the functionality of the visualization module of Abaqus/CAE.

```
beam_viewport = session.Viewport(name='Beam Results Viewport')
```

This statement uses the **Viewport()** method to create a **Viewport** object. The only required argument is **name** which is a String specifying the repository key. In this case we name it 'Beam Results Viewport'.

```
beam_Odb_Path = 'CantileverBeamJob.odb'
```

This statement assigns the name of the output database file to a variable for later use.

```
an_odb_object = session.openOdb(name=beam_Odb_Path)
```

This statement creates an **Odb** object by opening the output database whose path is provided as an argument, and assigns it to the variable **an_odb_object**. Note that we have not provided a complete path, only the file name, hence it will search for the file in the default Abaqus working directory. You may provide an absolute path if you are working with an output database file saved elsewhere on the hard drive.

```
beam_viewport.setValues(displayedObject=an_odb_object)
```

The statement uses the **setValues()** method to set the display to the selected output database. If you recall, this same method was used in the 'initialization block' (Section 4.3.1) of the script with **displayedObject=none** to blank the viewport. Just so you know, the above statement could have been written instead as

```
session.viewports['Beam Results Viewport']
                                  .setValues(displayedObject=an_odb_object)
```

The statement

```
beam_viewport.odbDisplay.display.setValues(plotState=(DEFORMED, ))
```

This statement changes the viewport display to the deformed beam by using the **setValues()** method and setting the plot state to the symbolic constant **DEFORMED**. For the record, the above statement could also have been written as

```
session.viewports['Beam Results Viewport'].odbDisplay
                                   .display.setValues(plotState=(DEFORMED, ))
```

4.4 What's Next?

In this chapter you worked through all the steps in the creation and setup of a finite element simulation in Abaqus using a Python script. Not only did you see the bigger picture, but you also examined individual statements and learnt of a number of new objects and methods that you will regularly encounter when scripting. In subsequent chapters we are going to look at many more examples, each of which we will perform tasks that weren't demonstrated in this one. But first, let's learn a little more Python syntax.

5

Python 102

5.1 Introduction

In Python 101, we covered many aspects of Python syntax. We spent a great deal of time understanding important concepts such as lists and tuples, and object oriented programming. That knowledge helped you understand the cantilever beam script. The example did not however use any conditional statements or any iterative loops. If…else… statements and for-loops are usually a major element in any sort of program you write, and you will need to use them in more complicated Python scripts as well. We'll cover them in this chapter.

This book assumes that you are familiar with at least one programming language, whether it be a full-fledged language like C++ or Java, or an engineering tool such as MATLAB. Hence the concepts of conditional statements and loops should not be new to you. This chapter aims only to familiarize you with the syntax of these constructs in Python.

5.1.1 If… elif … else statements

The **if** statement in Python is very similar to that used in other programming languages. It tests if a certain condition is true. If it is then it executes a statement or block of code.

If it is not true, Python checks to see if an else-if or else block is present. Else-if is written as **elif** in Python. **Elif** tests another condition whereas **else** does not test for any condition.

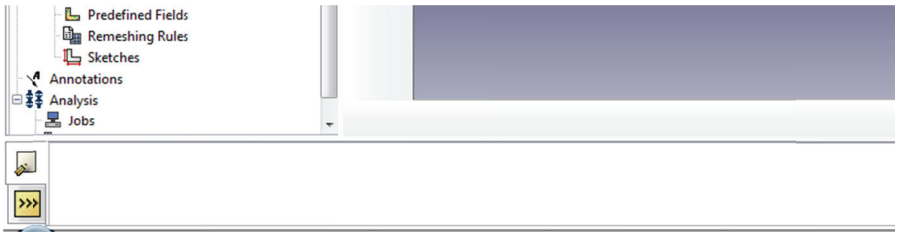
The syntax is a little different in Python. You do not put the **if** and **else** blocks of code within curly braces as you do in many other languages. In Python you indent the block instead. Also the colon ':' symbol is used. To indent the block is analogous to using

braces in other languages, if you don't do it you will get an error. The syntax looks something like this.

if a_certain_condition_is_true :
*   do this*
*   and this*
elif another_condition_is_true:
*   do this*
*   and this*
else:
*   do this*

Example

Open up Abaqus CAE. In the lower half of the window you see the message area. If you look to the left of the message area you see two tabs, one for **Message area** and the other for **Kernal Command Line Interface**.

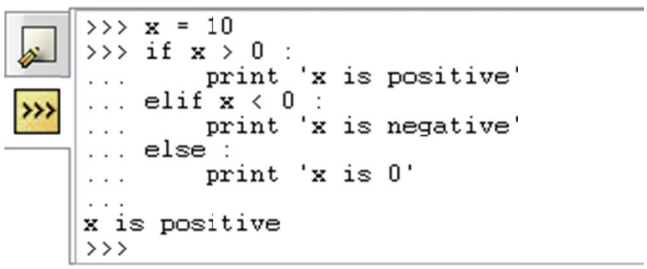


Click the second one. You see the kernel command prompt which is a ">>>" symbol.

Type the following lines, hitting the ENTER key on your keyboard after each.

```
X = 10
if x > 0 :
        print 'x is positive'
elif x < 0:
        print 'x is negative'
else :
        print 'x is 0'
```

Here is what you see



5.1.2 For loops

The **for** loop in Python is conceptually similar to that in other languages – it provides the ability to loop or iterate over a certain set of data. However its implementation is a little different in Python.

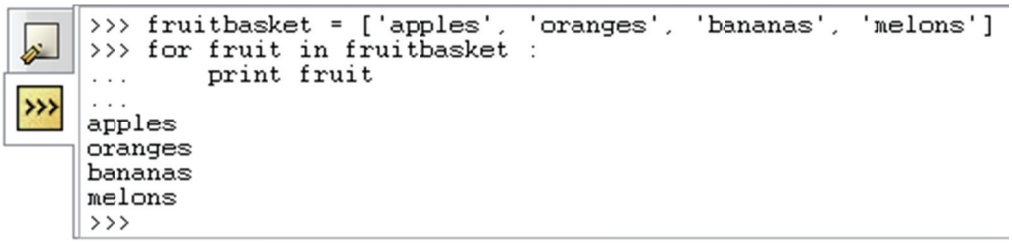
In C, C++, Java or MATLAB, you find yourself iterating either a fixed number of times by incrementing a variable every loop till it reaches a certain value, or until a condition is satisfied. In Python on the other hand, you create a sequence (a list or a string), and the for loop iterates over the items in that list (or characters in a string).

Example

Type the following statements in the Abaqus kernel command interface prompt

```
fruitbasket = [‘apples’, ‘oranges’, ‘bananas’, ‘melons’]
for fruit in fruitbasket :
        print fruit
```

Here is what you see:



In the above example, fruitbasket is a list consisting of a sequence of strings. With each iteration, the **for** loop takes an element (in this case a string) out of the list and assigns it

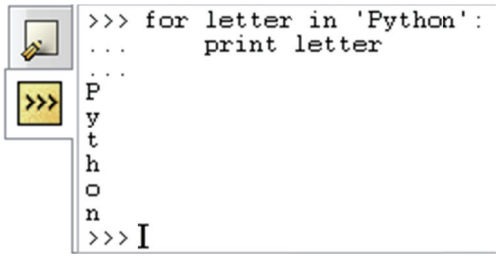
to the variable fruit. The print statement then prints it out on screen. Basically our **for** loop iterates 4 times.

Example

Type the following in the Abaqus kernel command interface prompt

```
for letter  in 'Python' :
        print letter
```

Here is what you see:



In the above example, 'Python' is a string, essentially a sequence of characters. With each iteration, the **for** loop takes an element (in this case a character) out of the String and assigns it to the variable letter. The print statement then prints it out on screen. So this for loop iterates 6 times.

This type of **for** loop is great for iterating through the elements of a list and performing an action on each one. Abaqus stores its repository keys in lists, hence it is easy to iterate through them using a **for** loop. This will be demonstrated in Chapter 8 while performing a dynamic, explicit truss analysis.

5.1.3 range() function

Sometimes you may wish to use a **for** loop to iterate a certain number of times, rather than loop through each element of a preexisting list. However the **for** loop can only operate on a sequence. A workaround is to generate a list for the task using the **range()** function.

The **range()** function generates a list which consists of arithmetic progressions. It can take one, two or three arguments. If one argument is provided, a list is generated starting

at 0, and ending at one integer less than the argument provided. It will naturally have the same number of elements as the value of the integer argument.

`range(5)` returns [0, 1, 2, 3, 4]

If two arguments are provided, the first one is treated as the beginning of the list, and the end of the list is one less than the second argument.

`range(5,9)` returns [5, 6, 7, 8]

If three arguments are provided, the first one is treated as the beginning of the list, and the end of the list is one less than the second one. However all elements in the list must be multiples of the third argument.

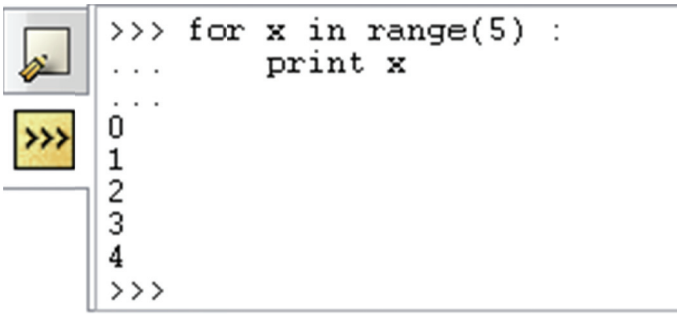
`range(2, 10, 3)` returns [3, 6, 9]

Using the **range()** function, you can specify a for loop to iterate a certain number of times.

Example

```
for x in range(5) :
        print x
```

Here is what you see:



The above for loop iterates 5 times. The range(5) statement returns a list [0, 1, 2, 3, 4] and the for loop iterates for each element (integer) in this list, assigning it to the variable x. The print statement prints this variable to the screen.

5.1.4 While-loops

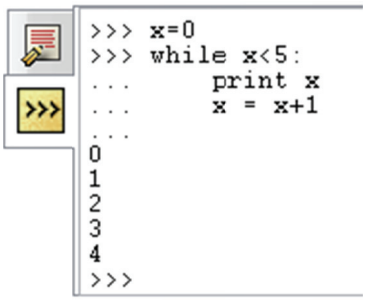
The **while** loop executes as long as a certain condition or expression returns true. It is similar to the **while** loop in other languages. The syntax is

```
while condition:
        do this
        and this
```

Example

```
x = 0
while x<5:
        print x
        x = x+1
```

Here is what you see



When the **while** loop is first encountered, x = 0, and the x < 5 condition is satisfied and the loop is executed. In each iteration the value of x is incremented by 1. When x = 5, the x<5 condition is no longer satisfied and control breaks out of the loop.

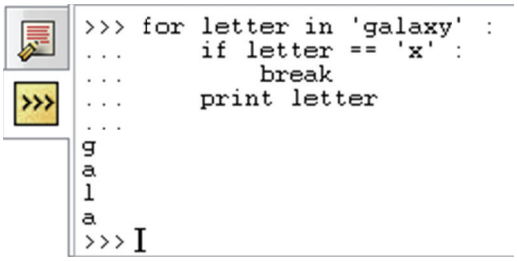
5.1.5 break and continue statements

The **break** statement allows program control to break out of a **for** loop or a **while** loop.

Example

```
for letter in ‘galaxy’ :
        if letter == ‘x’ :
                break
        print letter
```

Here is what you see:



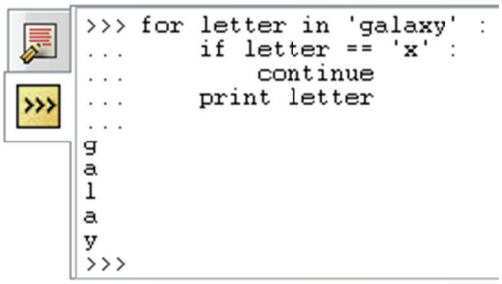
Each of the letters in the word galaxy are printed out turn by turn until the letter 'x' is reached. Since the **if** condition returns true, the **break** statement is encountered, and the program breaks out of the loop.

The **continue** statement on the other hand ends the current iteration without executing the remaining statements and begins the next iteration

Example

```
for letter in 'galaxy' :
        if letter == 'x' :
                continue
        print letter
```

Here is what you see:



Once again, each of the letters in the word galaxy are printed out turn by turn until the letter x is reached. Since the if-condition returns true, the **continue** statement is encountered. The current iteration is terminated before the print statement is executed, and the next iteration begins.

5.2 What's Next?

You now possess enough basic knowledge of Python syntax to proceed with scripting for Abaqus. The Python documentation, as well as a number of tutorials, are available at www.python.org if you wish to study the language further.

Before we start working with more examples, let's introduce you to some other important topics such as macros and replay files. Please proceed to the next chapter.

6

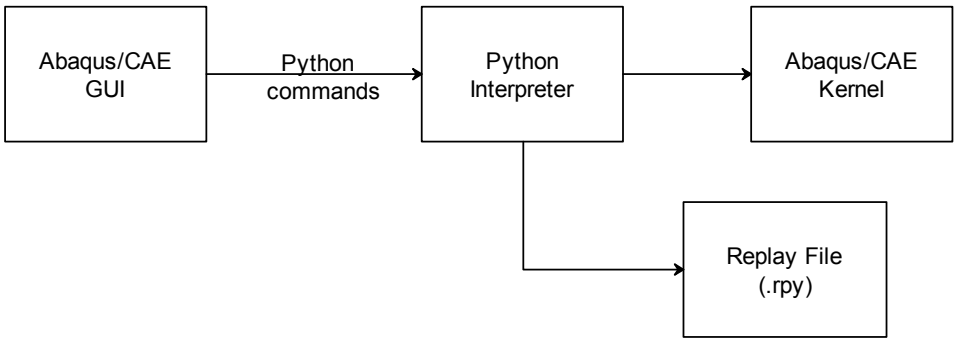
Replay files, Macros and IDEs

6.1 Introduction

The Abaqus Scripting Interface consists of thousands of commands and attributes separated into various Abaqus modules. It would be impossible for you to memorize all of these. Fortunately there is an easier way – replay files. In this chapter we'll talk about how you can use these. We'll also look at Macros, a feature provided by Abaqus, that makes it easy to create simple scripts without requiring any actual coding. And we'll get you hooked up with a good text editor to type your scripts through the rest of the book.

6.2 Replay Files

In Chapter 2, Section 2.2 (page 32), we talked about how Python fits into the bigger scheme of things. To summarize, when the user performs actions in the GUI (Abaqus/CAE), Python commands are generated which pass through the interpreter and are sent to the kernel. Fortunately for us, Abaqus keeps a record of these commands in the form of a replay file with the extension '.rpy'.



The replay file is written in the current work directory. The work directory is C:\Temp by default, and you can change it using **File > Set Work Directory..**

The easiest way to look up the necessary commands is to perform an action in Abaqus/CAE and then open up this replay file. If it is currently in use Abaqus may not let you open it; in this case right click on it and choose copy to create a copy of it in Windows Explorer that you can open.

NOTE: Abaqus Student Edition (current version at time of writing is 6.10-2) does not write replay files. This is one of its limitations. You need to be using the commercial or research editions of Abaqus for replay files to be written to the working directory. However if all you have is the student version, you can achieve the same thing with Macros. We will speak about these shortly. However I recommend you read the next section since everything with replay scripts applies to macros as well.

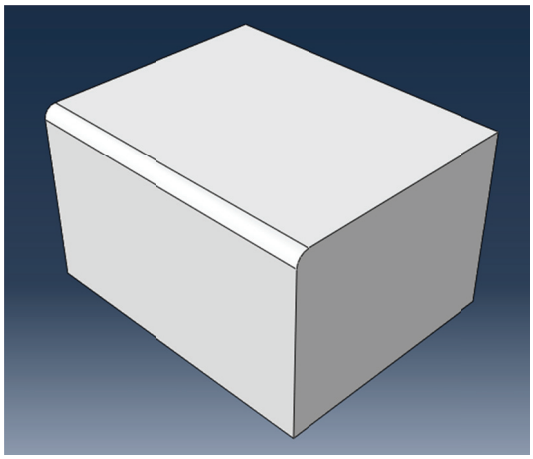
6.3 Example - Compare replay with a well written script

You will find that sometimes the replay file alone is exactly what you need for creating a script with minimal effort. For example if you open up a new moel in Abaqus/CAE, do a bunch of stuff, create parts, materials etc, you could copy all the statements from the replay file and save them in a .py file and use this in future to get back to the same point starting from a new model. It would be sort of like saving the .cae, except python scripts take up a lot less space and you can email them to people as text.

However if you are looking to work with the script, modify it, add iterative methods, or parametrize it, the form of the script in the replay file will most likely not be ideal. I'll demonstrate this with an example.

a. Start up Abaqus/CAE. If Abaqus is already open close it and reopen it as you start out with a blank replay file when you start a new Abaqus session.
b. Right click on **Model-1** in the model tree and choose **Rename**. Name it **Block Model**.
c. Double click on **Parts** in the model tree. You see the **Create Part** window.
d. Set the **Name** to **Block**, **modeling space** to **3D**, **type** to **Deformable**,**base feature shape** to **Solid**,**base feature type** to **Extrusion** and **approximate size** to **200**. Click **Continue**. You see the sketcher.
e. Choose the **Create Lines: Rectangle tool**. Click on the origin of the graph and then click anywhere in the top right quadrant to complete the rectangle.
f. Use the **Add Dimension** tool to give it a width of **25** and a height of **15**.
g. Click the red X to close the **Add Dimension** tool and then **Done** to exit the sketcher. You see the **Edit Base Extrusion** dialog box

h. Give the extrusion a **depth** of **20**. Click **OK**. You see the block in the viewport.
i. Choose the **Create Round or Fillet** tool. Click on the top left edge of the block to select it and choose **Done**
j. Give it a radius of **1**.
k. Click the red **X** to exit the **Create Round or Fillet** tool.



Now look in the Abaqus work directory which is C:\Temp by default or whatever you've set it to be. Open it in a text editor such as WordPad which comes with windows. (Notepad will not be good to view the replay file as a lot of the carriage returns are removed).

Here is what you will see (FYI I have modified the information in the top 3 lines):

```
# Abaqus/CAE Release 6.10-1 replay file
# Internal Version: xxxxxxxxxxxxxxxx
# Run by xxxxxx on Sat MonthDayxx:xx:xx 2011
#

# from driverUtils import executeOnCaeGraphicsStartup
# executeOnCaeGraphicsStartup()
#: Executing "onCaeGraphicsStartup()" in the site directory ...
from abaqus import *
from abaqusConstants import *
session.Viewport(name='Viewport: 1', origin=(0.0, 0.0), width=411.136439800262,
    height=212.019445240498)
session.viewports['Viewport: 1'].makeCurrent()
session.viewports['Viewport: 1'].maximize()
from caeModules import *
from driverUtils import executeOnCaeStartup
```

```
executeOnCaeStartup()
session.viewports['Viewport: 1'].partDisplay.geometryOptions.setValues(
    referenceRepresentation=ON)
mdb.models.changeKey(fromName='Model-1', toName='Block Model')
session.viewports['Viewport: 1'].setValues(displayedObject=None)
s = mdb.models['Block Model'].ConstrainedSketch(name='__profile__',
    sheetSize=200.0)
g, v, d, c = s.geometry, s.vertices, s.dimensions, s.constraints
s.setPrimaryObject(option=STANDALONE)
s.rectangle(point1=(0.0, 0.0), point2=(22.5, 12.5))
s.ObliqueDimension(vertex1=v[3], vertex2=v[0], textPoint=(6.54132556915283,
    -6.48623704910278), value=25.0)
s.ObliqueDimension(vertex1=v[0], vertex2=v[1], textPoint=(-8.33698463439941,
    4.81651592254639), value=15.0)
p = mdb.models['Block Model'].Part(name='Part-1', dimensionality=THREE_D,
    type=DEFORMABLE_BODY)
p = mdb.models['Block Model'].parts['Part-1']
p.BaseSolidExtrude(sketch=s, depth=20.0)
s.unsetPrimaryObject()
p = mdb.models['Block Model'].parts['Part-1']
session.viewports['Viewport: 1'].setValues(displayedObject=p)
del mdb.models['Block Model'].sketches['__profile__']
p = mdb.models['Block Model'].parts['Part-1']
e = p.edges
p.Round(radius=1.0, edgeList=(e[4], ))
```

As you can see, Abaqus has been recording everything you did in CAE in the replay file from the moment the software started up.

You see some statements that you would normally include in all scripts such as

```
from abaqus import *
from abaqusConstants import *
```

But you would be unlikely to write statements such as

```
session.Viewport(name='Viewport: 1', origin=(0.0, 0.0), width=411.136439800262,
    height=212.019445240498)
session.viewports['Viewport: 1'].makeCurrent()
session.viewports['Viewport: 1'].maximize()
from caeModules import *
from driverUtils import executeOnCaeStartup
executeOnCaeStartup()
```

in your script since you probably don't want your script to change the size of the viewport that it is run in, nor are you likely to want to run a startup script.

The remaining statements are the meat of the script. They rename the model, draw the sketch and create the part, and fillet it. However they are written in a very literal sense. For example, the **ObliqueDimensions()** command is used to dimension the edges of the rectangle. When you are using a script you are more likely to enter in the exact coordinates in the **rectangle()** method as **point1** and **point2** as we did in the cantilever beam example.

In addition the statements dealing with the edge round

```
e = p.edges
p.Round(radius=1.0, edgeList=(e[4], ))
```

appear to assign all the edges of the block to a variable ‘e’, and then Abaqus refers to the desired edge as e[4] which makes sense to it internally as it stores each of the **Edge** objects in a certain order; but this does not make any sense to a human.

Here is what this same script would look like if I wrote it.

```
# ******************************************************************************
# Create a block with a rounded edge

# Created for the book "Python Scripts for Abaqus - Learn by Example"
# Author: Gautam Puri
# ******************************************************************************

from abaqus import *
from abaqusConstants import *

# ------------------------------------------------
# Create the model (or more accurately, rename the existing one)

mdb.models.changeKey(fromName='Model-1', toName='Block Model')
blockModel = mdb.models['Block Model']

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ------------------------------------------------------------------
# Create the part

import sketch
import part

# a) Sketch the block cross section using the rectangle tool
blockProfileSketch = blockModel.ConstrainedSketch(name='Block CS Profile',
                                                  sheetSize=200)
blockProfileSketch.rectangle(point1=(0.0,0.0), point2=(25.0,15.0))
```

```
# b) Create a 3D deformable part named "Block" by extruding the sketch
blockPart=blockModel.Part(name='Block', dimensionality=THREE_D,
type=DEFORMABLE_BODY)
blockPart.BaseSolidExtrude(sketch=blockProfileSketch, depth=20)

# ----------------------------------------------------------------------
# Round the edge

edge_for_round = blockPart.edges.findAt((12.5, 15.0, 20.0), )
blockPart.Round(radius=1.0, edgeList=(edge_for_round, ))
```

The first thing you notice is how much more readable this script is. Secondly (and more importantly), we do not refer to internal edge or vertex lists. The statements for rounding the edge are

```
edge_for_round = blockPart.edges.findAt((12.5, 15.0, 20.0), )
blockPart.Round(radius=1.0, edgeList=(edge_for_round, ))
```

The **findAt()** method refers to coordinates that we can visualize by scribbling the block on a piece of paper. If you decided you wanted to round another edge in a second iteration of the analysis, you could change the coordinates right here and rerun the script. The replay file script on the other hand cannot be modified, since you wouldn't know what to change **e[4]** to since we do not know the sequence of Abaqus's internal edge list.

So you see that the replay file is useful only if you want to exactly replay what was done in Abaqus. However it requires some work to modify it for any other use. As it gets longer it will require too many major changes to be worth the effort.

However having a replay file helps you write your own script. You can see that the major methods used were the same in the replay script and the one I wrote. These include **changeKey()**, **ConstrainedSketch()**, **rectangle()**, **BaseSolidExtrude()** and **Round()**. By performing a task in Abaqus/CAE and looking at the replay file we very quickly know the names of the methods that need to be used and what arguments they require. While it is easy to remember a name like **Round()**, you are unlikely to remember the names of the thousands of other methods available through the Abaqus Scripting Interface. The replay file will tell you at a glance the names of the methods you need, and you can then look these up in the Abaqus Scripting Reference Manual to understand and use them.

Note also that my code is very similar to that used in the Cantilever Beam example. I have infact copied and pasted that code here, and modified it using some help from the replay file. The fastest way to write Python scripts is to reuse code where possible,

modify it suitably, and find out what new methods are required by performing the required task in Abaqus/CAE and reading the replay file. The only place you can't really do this is when dealing with output databases, but we'll get to ODB object model interrogation (after a few hundred pages) and teach you what you need to know then.

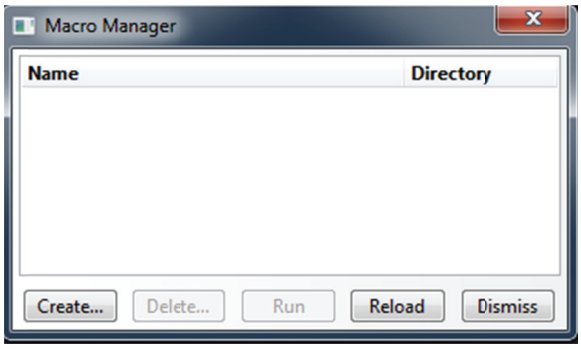
6.4 Macros

Macros are similar to replay files. The difference between them is that the replay file starts at the beginning of your Abaqus session and is continuously updated until you close Abaqus/CAE. In addition it can only be saved by making a copy of the .rpy file in Windows Explorer otherwise it will get overwritten during your next session. Macros on the other hand allow you to define at what point the replay data should start getting logged, and when it should stop. In addition you can give the replay data a name and call it later from within Abaqus. The statements in it will be the same as those in the .rpy file, except you won't have to search through hundreds of lines of other replay statements to find the few you need.

Macros are stored in a file called 'abaqusMacros.py'. Abaqus stores each macro within a function with the name you assign to the macro.

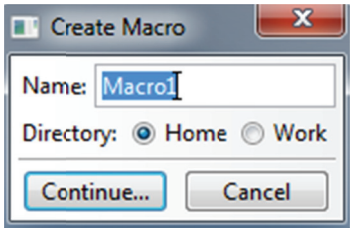
Let's demonstrate this:

Start Abaqus/CAE (or open a new model in Abaqus/CAE). Go to **File > Macro Manager**.

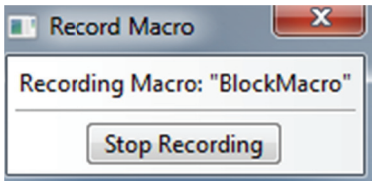


You see the **Macro Manager** dialog box as shown in the figure.

Click on **Create**. You see the **Create Macro** dialog box.



Type in a **name** for the macro such as **BlockMacro**. It needs to be one word as you cannot have a space in a macro name. This is because the name of the macro will be the name of the function in the abaqusMacros.py file and function names cannot have spaces. Change the directory to **Work** so that the macro is saved in the Abaqus work directory. Click **Continue**.



Abaqus begins recording the macro.

Repeat all the steps described in the previous section to rename the model, create the part 'Block' and round the edge. Then click **Stop Recording**.

You see BlockMacro appear in the list in the Macro Manager. As you create more macros they will appear here.

Open 'abaqusMacros.py' in the work directory. Here's what the contents will look like:

```
# Do not delete the following import lines
from abaqus import *
from abaqusConstants import *
import __main__

def BlockMacro():
    import section
    import regionToolset
    import displayGroupMdbToolset as dgm
    import part
    import material
    import assembly
    import step
```

```
import interaction
import load
import mesh
import job
import sketch
import visualization
import xyPlot
import displayGroupOdbToolset as dgo
import connectorBehavior
mdb.models.changeKey(fromName='Model-1', toName='Block Model')
session.viewports['Viewport: 1'].setValues(displayedObject=None)
s1 = mdb.models['Block Model'].ConstrainedSketch(name='__profile__',
    sheetSize=200.0)
g, v, d, c = s1.geometry, s1.vertices, s1.dimensions, s1.constraints
s1.setPrimaryObject(option=STANDALONE)
s1.rectangle(point1=(0.0, 0.0), point2=(22.5, 13.75))
s1.ObliqueDimension(vertex1=v[3], vertex2=v[0], textPoint=(16.4174423217773,
    -4.17431116104126), value=25.0)
s1.ObliqueDimension(vertex1=v[0], vertex2=v[1], textPoint=(-5.90002059936523,
    7.25688123703003), value=15.0)
p = mdb.models['Block Model'].Part(name='Block', dimensionality=THREE_D,
    type=DEFORMABLE_BODY)
p = mdb.models['Block Model'].parts['Block']
p.BaseSolidExtrude(sketch=s1, depth=20.0)
s1.unsetPrimaryObject()
p = mdb.models['Block Model'].parts['Block']
session.viewports['Viewport: 1'].setValues(displayedObject=p)
del mdb.models['Block Model'].sketches['__profile__']
p = mdb.models['Block Model'].parts['Block']
e1 = p.edges
p.Round(radius=1.0, edgeList=(e1[4], ))
```

You notice that the name of our macro ‘BlockMacro’ is the name of the function (indicated by the **def** keyword). In addition there are a number of **import** statements to import all modules that might be required by almost any script. Other than that the statements are the same as the ones in the replay file. Essentially what Abaqus has done is given you the statements of the replay file that were written while the macro was recording.

You can run an existing macro from the **Macro Manager** by choosing it from the list and clicking **Run**. In our case this will only work in a new model because we rename ‘Model-1’ to ‘Block Model’. (If no ‘Model-1’ is present then you will get an error.) If you’d used the macro to do something like create a material, you could then run the macro inside any instance of Abaqus and it would create that material for you again.

You can see how macros help you perform a repetitive task without actually writing a single Python statement yourself. The added advantage is that users of Abaqus Student Edition can use this in place of the replay file which they do not have access to. In fact even if you're using the Research or Commercial editions of Abaqus, you may prefer to create a macro of a task you are trying to script in order to see which commands Abaqus/CAE uses as opposed to reading the replay file which will include everything from the moment your Abaqus session began.

6.5 IDEs and Text Editors

Python scripts are basically text files with a .py extension. This means you can write them in the most basic of text editors – Notepad – which ships with every version of Windows. However you are unlikely to enjoy this experience too much, especially since Python code needs to be indented. In addition notepad displays everything in one font color, including things like comments, function names and import statements. This makes everything harder to read, and also harder to debug. You might enjoy scripting with something a little more sophisticated.

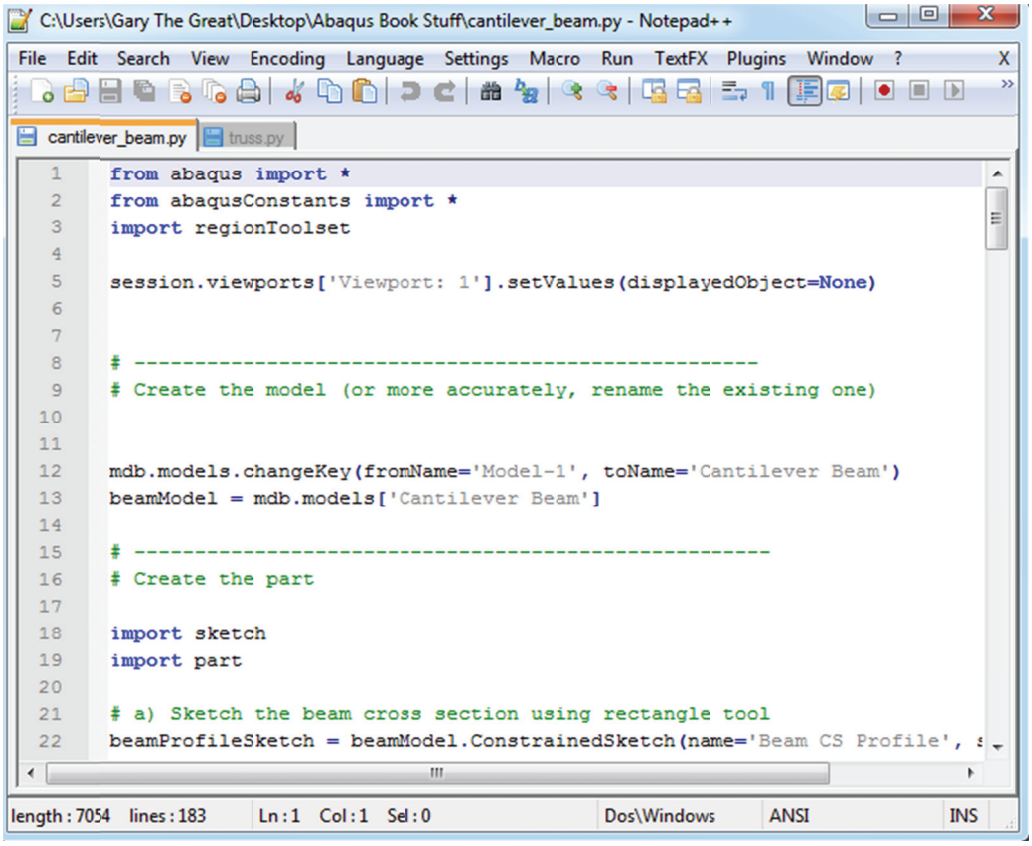
6.5.1 IDLE

IDLE is an IDE (integrated development environment) that is installed by default with any Python installation. Chances are it is already installed on your system if you look in the 'Start' menu in the Python application.

If you were programming in pure Python you could run your scripts directly from IDLE. However since you will be writing scripts for Abaqus, they would need to be run from within Abaqus/CAE (**File > Run Script**) or from the command line. You will essentially use IDLE as a text editor.

6.5.2 Notepad ++

Notepad++ is a free code editor. It is like an enhanced version of Notepad that is great for writing code. It has syntax highlighting and also displays line numbers next to statements which helps with debugging code. In addition you can have multiple files open in multiple tabs and switch between them easily. It supports a number of popular languages, including Python, and will choose the appropriate language and coloring based on the file extension.



All of the scripts for this book were written in Notepad++, it is my personal favorite. The website for Notepad++ (at the time of publication) is http://notepad-plus-plus.org/

6.5.3 Abaqus PDE

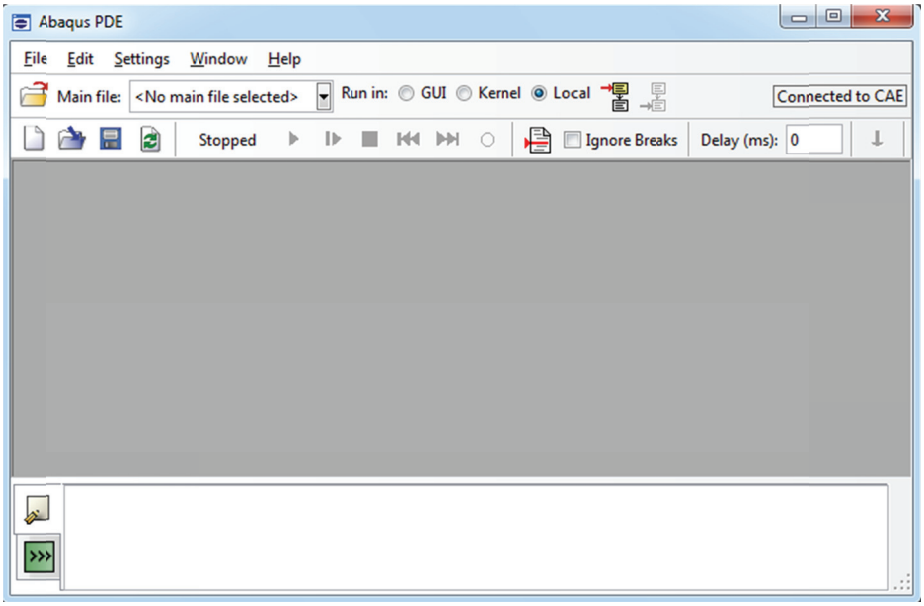
Abaqus Python Development Environment (PDE) is an application that comes bundled with Abaqus. It allows you to create and edit scripts, run then, and offers debugging features.

You can start Abaqus PDE from within Abaqus/CAE by going to **File > Abaqus PDE…** Alternatively you can start it by going to the system command prompt and typing (in Abaqus Student Edition version 6.10-2)

```
abq6102se -pde
```

You will need to change the 'abq6102se' to the command required to run your version of Abaqus (refer to Chapter 2 for details).

If you start Abaqus PDE from within Abaqus/CAE, it will be connected to CAE, as indicated by the words "Connected to CAE" displayed in the top left of the Abaqus PDE window (see figure). This means you will be using your Abaqus license tokens. If you run it from the command line however, Abaqus PDE will not be connected to CAE.



Abaqus PDE gives you the option to run the script in 3 modes – 'GUI', 'Kernel' and 'Local' in the toolbar (see figure). You choose the correct one depending on whether the scripts should run in Abaqus/CAE GUI, the Abaqus/CAE kernel or locally. By default .guiLog scripts run in GUI, and .py scripts run in the kernel.

What are .guiLog scripts? These are similar to macros, in the sense that you can perform some tasks in the GUI and a Python script will be written recording this. However .guiLog scripts describe the activity of the user in the GUI, which buttons were clicked and so on, whereas .py scripts record the Python commands called. So for example, when you close a dialog box, a .guiLog script records the fact that you clicked on a certain button, whereas a .py script records which function was called depending on the options you checked off in the dialog box.

This may be better understood with a demonstration. Open a new file in Abaqus PDE (**File > New Model Database > With Standard/Explicit Model**). Click the **Start Recording** button in the toolbar which appears as a red circle. Repeat all the steps from the previous section to rename the model, create a block and round an edge. Then click the **Stop Program** button represented by the solid square.

```
from abaqusTester import *
import abaqusGui
selectTreeListItem('Model Tree', ('Model Database','Models','Model-1'), 0)
showTreeListContextMenu('Model Tree')
selectMenuItem('Model Tree Menu + Rename')
setTextFieldValue('Rename Model + Rename To', 'Block Model')
pressButton('Rename Model + Ok')
selectTreeListItem('Model Tree', ('Model Database','Models','Block Model','Parts'),
0)
doubleClickTreeListItem('Model Tree', ('Model Database','Models','Block
Model','Parts'), 0)
setTextFieldValue('prtG_PartCreateDB + Create', 'Block Part')
pressButton('prtG_PartCreateDB + Continue')
pressButton('Sketcher GeomToolbox + Rectangle')
clickInViewport('Viewport: 1', (0.256754, -0.321101), 0.728166, LEFT_BUTTON)
clickInViewport('Viewport: 1', (27.216, 17.1468), 0.728166, LEFT_BUTTON)
pressButton('Sketcher ConsToolbox + Add Dimension')
clickInViewport('Viewport: 1', (5.00671, -0.0642202), 0.728166, LEFT_BUTTON)
clickInViewport('Viewport: 1', (8.21614, -8.15596), 0.728166, LEFT_BUTTON)
commitTextFieldValue('skcK_DimensionProcedure + New Dimension', '25')
clickInViewport('Viewport: 1', (-0.513509, 4.55963), 0.728166, LEFT_BUTTON)
clickInViewport('Viewport: 1', (-6.54723, 4.55963), 0.728166, LEFT_BUTTON)
commitTextFieldValue('skcK_DimensionProcedure + New Dimension', '15')
pressButton('Procedure + Cancel')
pressButton('prtK_NewPartProc + Done')
pressButton('prtG_ExtrudeFeatureDB + Ok')
pressFlyoutItem('Create Blend Flyout + Round/Fillet')
clickInViewport('Viewport: 1', (-0.112969, 0.0541739), 0.0044191, LEFT_BUTTON)
pressButton('prtK_BlendRoundProc + Done')
commitTextFieldValue('prtK_BlendRoundProc + Radius', '1.0')
pressButton('Procedure + Cancel')
```

You will notice that as you were working in the GUI, the .guiLog was storing a log of everything you did in the GUI. It is evident that this log is of a different nature compared to a script. It records information such as which button you clicked, where in the viewport you clicked, and even trivial things like clicking the ‘cancel procedure’ red X.

Let's see how this guiLog can be used. Create a new model in Abaqus by going to **File > New Model Database > With Standard/Explicit Model**. Leave the .guiLog file open in Abaqus PDE

Click the 'Play' button represented by the solid triangle. You will see that each of the lines in the .guiLog is highlighted one by one. At the same time, in the Abaqus/CAE window, you see the corresponding task being performed. It is almost like you are watching the person who created the guiLog at work except that you do not see their mouse cursor moving about. You may find it useful to pass a .guiLog file along to coworkers to demonstrate how you performed a task in the GUI.

At the bottom of the Abaqus PDE window, you see a message area and a command line interface similar to the one you see in Abaqus/CAE. The difference is that this is a GUI Command Line Interface whereas the one in Abaqus/CAE is a Kernel Command Line Interface. You will understand the difference between the two when we cover GUI customization in the last few chapters of the book. For now just know that a GUI API can be called from here, so you could for instance check the functionality of a dialog box.

Abaqus PDE has a number of debugging features. You can use the '**Set/Clear Breakpoint at cursor location**' tool to set a breakpoint at any statement (does not include comments or empty lines) and the statements before that point will be executed. You can then choose to contine after a breakpoint if you wish.

You can access the Abaqus PDE debugger using **Window > Debugger**. The debugger is displayed between the Abaqus PDE main window and the message area. You can display the watch list by clicking on 'show watch'. This allows you to watch the value of variables as the script executes. To add a variable to the watch list right click on it in the main window and select **Add Watch: (variable name).** This could be very useful for debugging purposes. Then again in Python it is quite common to debug code using 'Print' statements so go with your preference.

6.5.4 Other options

A free IDE popular in the Python world is PythonWin. Some individuals prefer this to IDLE. Another popular text editor is TextPad, which is quite similar to Notepad++. However this is not currently free but I believe you can try a fully functional evaluation version. A Google search will reveal many more options.

6.6 What's Next?

You will be relying heavily on replay files or macros when writing scripts, and you now understand how these work. Hopefully you've also decided on an IDE or text editor to use for subsequent examples.

You now have a basic knowledge of the Python programming language and an understanding of how to write scripts for Abaqus. You also know about replay files and macros. It is time to proceed to Part 2 of this book.

PART 2 – LEARN BY EXAMPLE

We shall now begin scripting in earnest. Every chapter in Part 2 is made up of one example. Each example introduces new topics and concepts. The first few examples/chapters create simple single run simulations. Subsequent chapters delve into topics of optimization, parameterization, output database processing and job monitoring.

For each example, the steps to perform the study in Abaqus/CAE are described. This is to ensure that you know how to run the simulation in the GUI before you script it. Instead of reading the procedure you may watch the videos on the book website. Following the CAE procedure is the corresponding script, and line-by-line explanation.

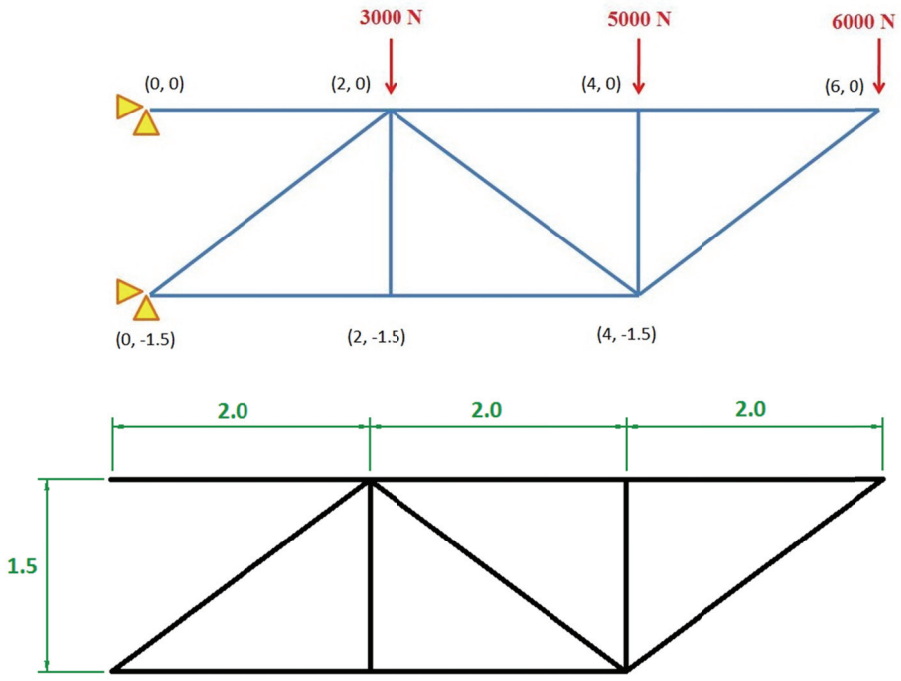
You don't necessarily need to read all of these chapters. However each of them demonstrates different tasks and if something is repeated the previous occurrence will be referenced. It might help to skim through each example and form a general idea of what each script does, so that you know where to find reusable code when writing your own scripts.

7

Static Analysis of a Loaded Truss

7.1 Introduction

In this chapter we will write a script to perform a static analysis on a truss. The problem is displayed in the figure. One end of the truss is fixed to a wall while the other end is free. Concentrated forces of 3000 N, 5000 N and 6000 N are applied to the nodes of the truss in the –Y direction.



(Dimensions are in meters)

In this example the following tasks will be demonstrated first using Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Create a static, general step
- Request field outputs
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job
- Plot overlaid deformed and undeformed results and display node numbers on plot
- Plot field outputs

The new topics covered are:

- Model / Preprocessing
  - Work in 2D
  - Create wire features
  - Create sections of type 'truss' and specify cross sectional areas
  - Use truss elements (with pin joints)
  - Use concentrated force loads
- Results / Post-processing
  - Allow multiple plot states (both deformed and undeformed plots overlaid)
  - Use Common Plot Options -> Show Node Labels
  - Display field output as color contours

7.2 Procedure in GUI

You can perform the simulation in Abaqus/CAE by following the steps listed below. You can either read through these, or watch the video demonstrating the process on the book website.

1. Rename **Model-1** to **Truss Structure**
   a. Right-click on Model-1 in Model Database
   b. Choose **Rename..**
   c. Change name to **Truss Structure**
2. Create the part
   a. Double-click on **Parts** in Model Database. **Create Part** window is displayed.
   b. Set **Name** to **Truss**
   c. Set **Modeling Space** to **2D Planar**
   d. Set **Type** to **Deformable**
   e. Set **Base Feature** to **Wire**
   f. Set **Approximate Size** to **10**
   g. Click **OK**. You will enter Sketcher mode.
3. Sketch the truss
   a. Use the **Create Lines:Connected**tool to draw the profile of the truss
   b. Split the lines using the **Split** tool
   c. Use **Add Constraints > Equal Length** tool to set the lengths of the required truss elements to be equal
   d. Use the **Add Dimension** tool to set the length of the horizontal elements to 2 m and the length of the vertical elements to 1.5 m.
   e. Click **Done** to exit the sketcher.
4. Create the material
   a. Double-click on **Materials** in the Model Database. **Edit Material** window is displayed
   b. Set **Name** to **AISI 1005 Steel**
   c. Select **General > Density**. Set **Mass Density** to **7872** (which is 7.872 g/cc)
   d. Select **Mechanical > Elasticity > Elastic**. Set **Young's Modulus** to **200E9** (which is 200 GPa) and **Poisson's Ratio** to **0.29**.
5. Assign sections
   a. Double-click on **Sections** in the Model Database. **Create Section** window is displayed
   b. Set **Name** to **Truss Section**
   c. Set **Category** to **Beam**
   d. Set **Type** to **Truss**
   e. Click **Continue…** The **Edit Section** window is displayed.
   f. In the **Basic** tab, set **Material** to **the AISI 1005 Steel** which was defined in the create material step.

g. Set **Cross-sectional Area** to **3.14E-4**
h. Click **OK**.

6. Assign the section to the truss
   a. Expand the **Parts** container in the Model Database. Expand the part **Truss**.
   b. Double-click on **Section Assignments**
   c. You see the message **Select the regions to be assigned a section** displayed below the viewport
   d. Click and drag with the mouse to select the entire truss.
   e. Click **Done**. The **Edit Section Assignment** window is displayed.
   f. Set **Section** to **Truss Section**.
   g. Click **OK**.
   h. Click **Done**.
7. Create the Assembly
   a. Double-click on **Assembly** in the Model Database. The viewport changes to the **Assembly Module**.
   b. Expand the **Assembly** container.
   c. Double-click on **Instances**. The **Create Instance** window is displayed.
   d. Set **Parts** to **Truss**
   e. Set **Instance Type** to **Dependent (mesh on part)**
   f. Click **OK**.
8. Create Steps
   a. Double-click on **Steps** in the Model Database. The **Create Step** window is displayed.
   b. Set **Name** to **Loading Step**
   c. Set **Insert New Step After** to **Initial**
   d. Set **Procedure Type** to **General > Static, General**
   e. Click **Continue..** The **Edit Step** window is displayed
   f. In the **Basic** tab, set **Description** to **Loads are applied to the truss in this step**.
   g. Click **OK**.
9. Request Field Outputs
   a. Expand **the Field Output Requests** container in the Model Database.
   b. Right-click on **F-Output-1** and choose Rename…
   c. Change the name to **Selected Field Outputs**
   d. Double-click on **Selected Field Outputs** in the Model Database. The **Edit Field Output Request** window is displayed.

e. Select the desired variables by checking them off in the **Output Variables** list. The variables we want are **S (stress components and invariants)**, **U (translations and rotations)**, **RF (reaction forces and moments)**, and **CF (concentrated forces and moments)**. Uncheck the rest. You will notice that the text box above the output variable list displays **S,U,RF,CF**
f. Click **OK**.

10. Assign Loads
    a. Double-click on **Loads** in the Model Database. The **Create Load** window is displayed
    b. Set **Name** to **Force1**
    c. Set **Step** to **Loading Step**
    d. Set **Category** to **Mechanical**
    e. Set **Type for Selected Step** to **Concentrated Force**
    f. Click **Continue…**
    g. You see the message **Select points for the load displayed below the viewport**
    h. Select the upper left node by clicking on it
    i. Click **Done**. The **Edit Load** window is displayed
    j. Set **CF2** to **-3000** to apply a 3000 N force in downward (negative Y) direction
    k. Click **OK**
    l. You will see the force displayed with an arrow in the viewport on the selected node
    m. Repeat steps a-l two more times, once each for the upper middle and upper right node. Name the forces **Force2** and **Force3**, and set them to **-5000** and **-6000** respectively.
11. Apply boundary conditions
    a. Double-click on **BCs** in the Model Database. The **Create Boundary Condition** window is displayed
    b. Set **Name** to **Pin1**
    c. Set **Step** to **Initial**
    d. Set **Category** to **Mechanical**
    e. Set **Types for Selected Step** to **Displacement/Rotation**
    f. Click **Continue…**
    g. You see the message **Select regions for the boundary condition** displayed below the viewport

h. Select the two nodes on the extreme left. You can press the "Shift" key on your keyboard to select both at the same time.
i. Click **Done**. The **Edit Boundary Condition** window is displayed.
j. Check off **U1** and **U2**. This will create a pin joint which does not allow translation but permits rotation.
k. Click **OK**.

12. Create the mesh
    a. Expand the **Parts** container in the Model Database.
    b. Expand **Truss**
    c. Double-click on **Mesh (Empty)**. The viewport window changes to the **Mesh module** and the tools in the toolbar are now meshing tools.
    d. Using the menu bar click on **Mesh > Element Type …**
    e. You see the message **Select the regions to be assigned element types** displayed below the viewport
    f. Click and drag using your mouse to select the entire truss.
    g. Click **Done**. The **Element Type** window is displayed.
    h. Set **Element Library** to **Standard**
    i. Set **Geometric Order** to **Linear**
    j. Set **Family** to **Truss**
    k. You will notice the message **T2D2: A 2-node linear 2-D truss**
    l. Click **OK**
    m. Click **Done**
    n. Using the menu bar lick on **Seed > Edge by Number**
    o. You see the message **Select the regions to be assigned local seeds** displayed below the viewport
    p. Click and drag using your mouse to select the entire truss
    q. Click **Done**.
    r. You see the prompt **Number of elements along the edges** displayed below the viewport.
    s. Set it to **1** and press the "Enter" key on your keyboard
    t. Click **Done**
    u. Using the menu bar click on **Mesh > Part**
    v. You see the prompt **OK to mesh the part?** displayed below the viewport
    w. Click **Yes**
13. Create and submit the job

a. Double-click on **Jobs** in the Model Database. The **Create Job** window is displayed
b. Set **Name** to **TrussAnalysisJob**
c. Set **Source** to **Model**
d. Select **Truss Structure** (it is the only option displayed)
e. Click **Continue..** The **Edit Job** window is displayed
f. Set **Description** to **Analysis of truss under concentrated loads**
g. Set **Job Type** to **Full Analysis**.
h. Leave all other options at defaults
i. Click **OK**
j. Expand the**Jobs** container in the Model Database
k. Right-click on **TrussAnalysisJob** and choose **Submit**. This will run the simulation. You will see the following messages in the message window:
The job input file "TrussAnalysisJob.inp" has been submitted for analysis.
Job TrussAnalysisJob: Analysis Input File Processor completed successfully
Job TrussAnalysisJob: Abaqus/Standard completed successfully
Job TrussAnalysisJob completed successfully

14. Plot results deformed and undeformed
a. Right-click on **TrussAnalysisJob (Completed)** in the Model Database. Choose **Results**.The viewport changes to the **Visualization** module.
b. In the toolbar click the **Plot Undeformed Shape** tool. The truss is displayed in its undeformed state.
c. In the toolbar click the **Plot Deformed Shape** tool. The truss is displayed in its deformed state.
d. In the toolbar click the **Allow Multiple Plot States** tool. Then click the **Plot Undeformed Shape** tool. Both undeformed and deformed shapes are now visible superimposed on one another.
e. Click again on the **Allow Multiple Plot States** tool to disallow this feature. Click on **Plot Deformed Shape** to have the deformed state displayed once again in the viewport.
f. In the toolbar click the **Common Options** tool. The **Common Plot Options** window is displayed.
g. In the **Labels** tab check **Show node labels**
h. Click **OK**. The nodes are now numbered on the truss in the viewport.

15. Plot Field Outputs
    a. Using the menu bar click on **Result > Field Output..**. The **Field Output** window is displayed.
    b. In the **Output Variable** list select **U** which has the description **Spatial displacement at nodes**. In the **Invariant** list **Magnitude** is displayed. In the **Components** list **U1** and **U2** are displayed
    c. In the I**nvariant** list select **Magnitude**. Click **Apply**. You might see the **Select Plot State** window with the message **The field output variable has been set, but it will not affect the current Display Group instance unless a different plot state is selected below.** For the **Plot state** select **Contour** and click **OK**.
    d. Click **OK** to close the **Field Output** window. You notice in the viewport a color contour has been applied on the truss with a legend indicating the **U** magnitude.
    e. Once again, using the menu bar click on **Result > Field Output...** The **Field Output** window is displayed.
    f. In the **Output Variab**le list select **U** which has the description **Spatial displacement at nodes**.
    g. In the **Component** list select **U1**.
    h. Click **OK**. The visualization updates to display **U1** which is displacement in the X direction.

7.3 Python Script

The following Python script replicates the above procedure for the static analysis of the truss. You can find it in the source code accompanying the book in **truss.py**. You can run it by opening a new model in Abaqus/CAE (**File > New Model database > With Standard/Explicit Model**) and running it with **File > Run Script…**

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)

# ------------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Truss Structure')
trussModel = mdb.models['Truss Structure']
```

```
# ------------------------------------------------------------------------
# Create the part

import sketch
import part

trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
trussSketch.Line(point1=(0, 0), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, 0))
trussSketch.Line(point1=(4, 0), point2=(6, 0))
trussSketch.Line(point1=(0, -1.5), point2=(2,-1.5))
trussSketch.Line(point1=(2, -1.5), point2=(4,-1.5))
trussSketch.Line(point1=(0, -1.5), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, -1.5))
trussSketch.Line(point1=(4, -1.5), point2=(6, 0))
trussSketch.Line(point1=(2, 0), point2=(2, -1.5))
trussSketch.Line(point1=(4, 0), point2=(4, -1.5))

trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
trussPart.BaseWire(sketch=trussSketch)

# ------------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
trussMaterial = trussModel.Material(name='AISI 1005 Steel')
trussMaterial.Density(table=((7872, ),       ))
trussMaterial.Elastic(table=((200E9, 0.29), ))

# ------------------------------------------------------------------------
# Create a section and assign the truss to it
import section

trussSection = trussModel.TrussSection(name='Truss Section',
                                       material='AISI 1005 Steel',
                                       area=3.14E-4)

edges_for_section_assignment = trussPart.edges.findAt(((1.0, 0.0, 0.0), ),
                                                      ((3.0, 0.0, 0.0), ),
                                                      ((5.0, 0.0, 0.0), ),
                                                      ((1.0, -1.5, 0.0), ),
                                                      ((3.0, -1.5, 0.0), ),
                                                      ((1.0, -0.75, 0.0), ),
                                                      ((3.0, -0.75, 0.0), ),
                                                      ((5.0, -0.75, 0.0), ),
                                                      ((2.0, -0.75, 0.0), ),
                                                      ((4.0, -0.75, 0.0), ))
```

```
truss_region = regionToolset.Region(edges=edges_for_section_assignment)
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')

# ------------------------------------------------------------------------
# Create the assembly

                                (Removed from Preview)


# ------------------------------------------------------------------------
# Create the step

                                (Removed from Preview)


# ------------------------------------------------------------------------
# Create the field output request


                                (Removed from Preview)


# ------------------------------------------------------------------------
# Create the history output request
# We want the defaults so we'll leave this section blank

# ------------------------------------------------------------------------
# Apply loads

                                (Removed from Preview)


# ------------------------------------------------------------------------
# Apply boundary conditions


                                (Removed from Preview)

# ------------------------------------------------------------------------
# Create the mesh


                                (Removed from Preview)

# ------------------------------------------------------------------------
# Create and run the job


                                (Removed from Preview)
```

```
# ------------------------------------------------------------------------
# Post processing

import visualization

truss_Odb_Path = 'TrussAnalysisJob.odb'
odb_object = session.openOdb(name=truss_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                          .setValues(plotState=(DEFORMED, ))

# Plot the deformed state of the truss
truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                                DEFORMED, ))
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)

# Plot the output variable U (spatial displacements at nodes) as its Magnitude
# invariant
# This is the equivalent of going to Report > Field Output and choosing to
# output U with Invariant: Magnitude
truss_displacements_magnitude_viewport= session \
                      .Viewport(name='Truss Displacements at Nodes (Magnitude)')
truss_displacements_magnitude_viewport.setValues(displayedObject=odb_object)
truss_displacements_magnitude_viewport.odbDisplay \
                                  .setPrimaryVariable(variableLabel='U',
                                                      outputPosition=NODAL,
                                                      refinement=(INVARIANT,
                                                                  'Magnitude'))
truss_displacements_magnitude_viewport.odbDisplay.display \
                                     .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_magnitude_viewport.setValues(width=250, height=160)
truss_displacements_magnitude_viewport.offset(20,-10)

# Plot the output variable U (spatial displacements at nodes) as its U1 component
# This is the equivalent of going to Report > Field Output and choosing to output
# U with Component: U1
truss_displacements_U1_viewport= session \
                    .Viewport(name='Truss Displacements at Nodes (U1 Component')
truss_displacements_U1_viewport.setValues(displayedObject=odb_object)
truss_displacements_U1_viewport.odbDisplay \
                           .setPrimaryVariable(variableLabel='U',
                                               outputPosition=NODAL,
                                               refinement=(COMPONENT, 'U1'))
truss_displacements_U1_viewport.odbDisplay.display \
                                     .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_U1_viewport.setValues(width=250, height=160)
truss_displacements_U1_viewport.offset(40,-20)
```

```
session.viewports['Viewport: 1'].sendToBack()
```

7.4 Examining the Script

Let's go through the entire script, statement by statement, and understand how it works.

7.4.1 Initialization (import required modules)

The block dealing with this initialization is

```
from abaqus import *
from abaqusConstants import *
import regionToolset

session.viewports['Viewport: 1'].setValues(displayedObject=None)
```

These statements are identical to those used in the Cantilever Beam example and were explained in section 4.3.1 on page59

7.4.2 Create the model

The following code block creates the model

```
# ------------------------------------------------------------------------
# Create the model

mdb.models.changeKey(fromName='Model-1', toName='Truss Structure')
trussModel = mdb.models['Truss Structure']
```

These statements rename the model from 'Model-1' to 'Truss Structure'. They are almost identical to those used in the Cantilever Beam example and were explained in section 4.3.2 on page 61.

7.4.3 Create the part

The following block creates the part

```
# ------------------------------------------------------------------------
# Create the part

import sketch
import part

trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
```

```
trussSketch.Line(point1=(0, 0), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, 0))
trussSketch.Line(point1=(4, 0), point2=(6, 0))
trussSketch.Line(point1=(0, -1.5), point2=(2,-1.5))
trussSketch.Line(point1=(2, -1.5), point2=(4,-1.5))
trussSketch.Line(point1=(0, -1.5), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, -1.5))
trussSketch.Line(point1=(4, -1.5), point2=(6, 0))
trussSketch.Line(point1=(2, 0), point2=(2, -1.5))
trussSketch.Line(point1=(4, 0), point2=(4, -1.5))

trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
trussPart.BaseWire(sketch=trussSketch)
```

```
import sketch
import part
```

These statements import the sketch and part modules into the script, thus providing access to the objects related to sketches and parts. They were explained in section 4.3.3 on page62.

```
trussSketch = trussModel.ConstrainedSketch(name='2D Truss Sketch', sheetSize=10.0)
```

This statement creates a constrained sketch object by calling the **ConstrainedSketch()** method of the **Model** object. This was explained in section 4.3.3 on page 63.

```
trussSketch.Line(point1=(0, 0), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, 0))
trussSketch.Line(point1=(4, 0), point2=(6, 0))
trussSketch.Line(point1=(0, -1.5), point2=(2,-1.5))
trussSketch.Line(point1=(2, -1.5), point2=(4,-1.5))
trussSketch.Line(point1=(0, -1.5), point2=(2, 0))
trussSketch.Line(point1=(2, 0), point2=(4, -1.5))
trussSketch.Line(point1=(4, -1.5), point2=(6, 0))
trussSketch.Line(point1=(2, 0), point2=(2, -1.5))
trussSketch.Line(point1=(4, 0), point2=(4, -1.5))
```

The statements use the **Line()** method of the **ConstrainedSketchGeometry** object. The **ConstrainedSketchGeometry** object stores the geometry of a sketch, such as lines, circles, arcs, and construction lines. The sketch module defines **ConstrainedSketchGeometry** objects. The first parameter **point1** is a pair of floats specifying the coordinates of the first endpoint of the line. The second parameter **point2** is a pair of floats specifying the coordinates of the second endpoint.

```
trussPart = trussModel.Part(name='Truss', dimensionality=TWO_D_PLANAR,
                            type=DEFORMABLE_BODY)
```

This statement creates a **Part** object and places it in the parts repository. The **name** of the part (its key in the repository) is set to 'Truss' and its **dimensionality** is set to a SymbolicConstant **TWO_D_PLANAR** which defines it to be a 2D part. It is defined to be of the type deformable body using the **DEFORMABLE_BODY** SymbolicConstant.

```
trussPart.BaseWire(sketch=trussSketch)
```

This statement calls the **BaseWire()** method which creates a Feature object by creating a planar wire from the ConstrainedSketch object **trussSketch** which is passed to it as an argument. Feature objects were explained in section 4.3.3 on page 64.

7.4.4 Define the materials

The following block of code creates the material for the simulation

```
# -----------------------------------------------------------------------
# Create material

import material

# Create material AISI 1005 Steel by assigning mass density, youngs modulus
# and poissons ratio
trussMaterial = trussModel.Material(name='AISI 1005 Steel')
trussMaterial.Density(table=((7872, ),       ))
trussMaterial.Elastic(table=((200E9, 0.29), ))
```

The statements are almost identical to those used in the Cantilever Beam example and were explained in section 4.3.4 on page 64.

7.4.5 Create sections and make section assignments

The following block creates the section and assigns it to the truss

```
# -----------------------------------------------------------------------
# Create a section and assign the truss to it
import section

trussSection = trussModel.TrussSection(name='Truss Section',
                                       material='AISI 1005 Steel',
                                       area=3.14E-4)

edges_for_section_assignment = trussPart.edges.findAt(((1.0, 0.0, 0.0), ),
                                                      ((3.0, 0.0, 0.0), ),
                                                      ((5.0, 0.0, 0.0), ),
                                                      ((1.0, -1.5, 0.0), ),
                                                      ((3.0, -1.5, 0.0), ),
```

```
                                                        ((1.0, -0.75, 0.0), ),
                                                        ((3.0, -0.75, 0.0), ),
                                                        ((5.0, -0.75, 0.0), ),
                                                        ((2.0, -0.75, 0.0), ),
                                                        ((4.0, -0.75, 0.0), ))

truss_region = regionToolset.Region(edges=edges_for_section_assignment)
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')
```

```
import section
```

This statement imports the section module making its properties and methods accessible to the script.

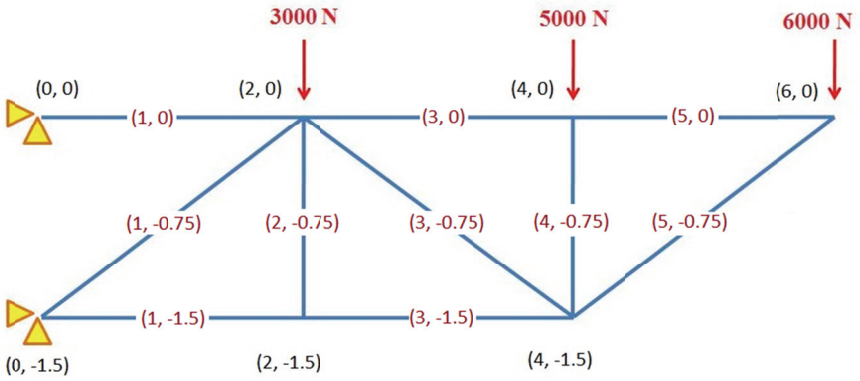
```
trussSection = trussModel.TrussSection(name='Truss Section',
                                       material='AISI 1005 Steel',
                                       area=3.14E-4)
```

This statement creates a **TrussSection** object using the **TrussSection()** method. The **TrussSection** object is derived from the **Section** object which is defined in the section module. The first parameter given to the method is a String for the name, which is used as the repository key. The second parameter is the material, which has been defined. Note that this material parameter must be a String, it cannot be a **material** object. That means we cannot say *material=trussMaterial* even though we had defined the **trussMaterial** variable earlier. 'AISI1005 Steel' on the other hand is a String, and it is the key assigned to that material in the **materials** repository. The third argument, area, is an optional one. It is a Float specifying the cross-sectional area of the truss members. Since our truss members have a radius of 1 cm (or 0.01 m), their cross-sectional area is 0.000314 m^2.

```
edges_for_section_assignment = trussPart.edges.findAt(((1.0, 0.0, 0.0), ),
                                                      ((3.0, 0.0, 0.0), ),
                                                      ((5.0, 0.0, 0.0), ),
                                                      ((1.0, -1.5, 0.0), ),
                                                      ((3.0, -1.5, 0.0), ),
                                                      ((1.0, -0.75, 0.0), ),
                                                      ((3.0, -0.75, 0.0), ),
                                                      ((5.0, -0.75, 0.0), ),
                                                      ((2.0, -0.75, 0.0), ),
                                                      ((4.0, -0.75, 0.0), ))
```

This statement uses the **findAt()** method to find any objects in the **EdgeArray** (basically edges) at the specified points or at a distance of less than 1E-6 from them. **trussPart** is the part, **trussPart.edges** exposes the **EdgeArray**, and **trussPart.edges.findAt()** finds the **edge** in the **EdgeArray**.

The coordinates used were obtained by drawing a rough sketch and determining the midpoints of each of the truss members. They are displayed in the figure below. Note that the Z coordinate was added when using the **findAt()** method. Being a 2D object the Z coordinate is 0.0 for all points.




```
truss_region = regionToolset.Region(edges=edges_for_section_assignment)
```

This statement creates a **Region** object using the **Region()** method. The **Region()** method has no required arguments, only optional ones such as **elements**, **nodes**, **vertices**, **edges**, **faces**, **cells** and a few more listed in the documentation. We use the **edges** argument, and assign it the edges obtained in the previous statement, which are the member elements of the truss.

The **Region** object itself was discussed in section 4.3.5 of the Cantilever Beam example on page 67. Note how the method used to create the region in this example is different from that used in the Cantilever Beam example. With the beam, a 3D object, we created **beam_region** with the statement *beam_region=(beamPart.cells,)* With the truss, a 2D planar object, we instead use the **Region()** method and passing the edges as arguments.

```
trussPart.SectionAssignment(region=truss_region, sectionName='Truss Section')
```

This statement creates a **SectionAssignment** object using the **SectionAssignment()** method. It is almost identical to the one used in the Cantilever Beam example, section 4.3.5 on page 67.The first parameter is the **Region** object created in the previous statement, and the second parameter is the name we wish to give the section, which is also its key in the sections repository.

7.4.6 Create an assembly

(Section removed from Preview)

7.4.7 Create steps

(Section removed from Preview)

7.4.8 Create and define field output requests

(Section removed from Preview)

7.4.9 Create and define history output requests

(Section removed from Preview)

7.4.10 Apply loads

(Section removed from Preview)

7.4.11 Apply boundary conditions

(Section removed from Preview)

7.4.12 Mesh

(Section removed from Preview)

7.4.13 Create and run the job

(Section removed from Preview)

7.4.14 Post processing – setting the viewport

The following code begins the post processing

```
# ------------------------------------------------------------------------
# Post processing

import visualization

truss_Odb_Path = 'TrussAnalysisJob.odb'
odb_object = session.openOdb(name=truss_Odb_Path)

session.viewports['Viewport: 1'].setValues(displayedObject=odb_object)
session.viewports['Viewport: 1'].odbDisplay.display \
                                            .setValues(plotState=(DEFORMED, ))
```

You have seen these statements used in the Cantilever Beam example. To refresh your memory refer back to section 0 on page 69.

7.4.15 Plot the deformed state and modify common options

The following post processing block plots the deformed state of the truss and enables node and element labels through the common options

```
# Plot the deformed state of the truss
truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                                DEFORMED, ))
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)
```

```
truss_deformed_viewport = session.Viewport(name='Truss in Deformed State')
truss_deformed_viewport.setValues(displayedObject=odb_object)
```

These 2 statements should look familiar to you. The first one creates a new **Viewport** object (a new window on your screen) called 'Truss in Deformed State'. The second statement assigns the output database of the simulation to the viewport.

```
truss_deformed_viewport.odbDisplay.display.setValues(plotState=(UNDEFORMED,
                                                                DEFORMED, ))
```

You have seen the **setValues()** method used in the Cantilever Beam example. The difference here is that two symbolic keywords **UNDEFORMED** and **DEFORMED** have been used together. This causes both to be displayed overlaid on one another in the viewport window.

```
truss_deformed_viewport.odbDisplay.commonOptions.setValues(nodeLabels=ON)
```

This statement is the equivalent of clicking on the Common Options tool in the viewport and checking off 'show node labels'. Notice how we have again used the **setValues()** method, just as in the last statement, but the arguments supplied to it are very different. The parameters of the **setValues()** method depend on the context you are using it in.

```
truss_deformed_viewport.odbDisplay.commonOptions.setValues(elemLabels=ON)
```

This statement is the equivalent of clicking on the Common Options tool in the viewport and checking off 'show element labels'.

```
truss_deformed_viewport.setValues(origin=(0.0, 0.0), width=250, height=160)
```

Once again we use the **setValues()** method on the **Viewport** object. This time we provide 3 optional arguments, the **origin** of the new viewport window, its **width** and its **height**.

7.4.16 Plot the field outputs

The following post processing block plots the field output variables

```
# Plot the output variable U (spatial displacements at nodes) as its Magnitude
# invariant
# This is the equivalent of going to Report > Field Output and choosing to
# output U with Invariant: Magnitude
truss_displacements_magnitude_viewport= session \
                      .Viewport(name='Truss Displacements at Nodes (Magnitude)')
truss_displacements_magnitude_viewport.setValues(displayedObject=odb_object)
truss_displacements_magnitude_viewport.odbDisplay \
                                    .setPrimaryVariable(variableLabel='U',
                                                        outputPosition=NODAL,
                                                        refinement=(INVARIANT,
                                                                    'Magnitude'))
truss_displacements_magnitude_viewport.odbDisplay.display \
                                       .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_magnitude_viewport.setValues(width=250, height=160)
truss_displacements_magnitude_viewport.offset(20,-10)

# Plot the output variable U (spatial displacements at nodes) as its U1 component
# This is the equivalent of going to Report > Field Output and choosing to output
# U with Component: U1
truss_displacements_U1_viewport= session \
                     .Viewport(name='Truss Displacements at Nodes (U1 Component')
truss_displacements_U1_viewport.setValues(displayedObject=odb_object)
truss_displacements_U1_viewport.odbDisplay \
                             .setPrimaryVariable(variableLabel='U',
                                                 outputPosition=NODAL,
                                                 refinement=(COMPONENT, 'U1'))
truss_displacements_U1_viewport.odbDisplay.display \
                                       .setValues(plotState=(CONTOURS_ON_DEF, ))
truss_displacements_U1_viewport.setValues(width=250, height=160)
truss_displacements_U1_viewport.offset(40,-20)

session.viewports['Viewport: 1'].sendToBack()
```

```
truss_displacements_magnitude_viewport= session \
                      .Viewport(name='Truss Displacements at Nodes (Magnitude)')
truss_displacements_magnitude_viewport.setValues(displayedObject=odb_object)
```

You are very familiar by now with the above 2 statements. We are creating a new viewport window called ‘Truss Displacements at Nodes (Magnitude)’ and setting it to draw its data from the output database file.

```
truss_displacements_magnitude_viewport.odbDisplay \
                                    .setPrimaryVariable(variableLabel='U',
                                                        outputPosition=NODAL,
                                                        refinement=(INVARIANT,
                                                                    'Magnitude'))
```

The **setPrimaryVariable()** method is used, which specifies the field output variable for which to obtain results from the output database. The first required argument **variableLabel** is a String specifying the field output variable we wish to plot. The second required argument, **outputPosition** requires a SymbolicConstant specifying the position from which to obtain data. One of the possible values is **NODAL**, which indicates we are drawing the data from a node. The documentation lists other possible values. The third argument is an optional one called **refinement**. It is only required if a refinement is available for the specified **variableLabel**, which is the case here. It must be a sequence of a SymbolicConstant and a String. We set the SymbolicConstant to **INVARIANT** and the String to 'Magnitude'.

```
truss_displacements_magnitude_viewport.odbDisplay.display \
                                        .setValues(plotState=(CONTOURS_ON_DEF, ))
```

You once again see the **setValues()** method being used on the **Display** object. Previously we set the **plotState** variable to the SymbolicConstants **DEFORMED** or **UNDEFORMED** (or both). In this situation we are setting the plot state to **CONTOURS_ON_DEF** which tells Abaqus to display the deformed state with a color contour of the specified variable/quantity (ie, U) displayed on it.

```
truss_displacements_magnitude_viewport.setValues(width=250, height=160)
```

Once again we use the **setValues()** method on the viewport and provide the optional width and height arguments to set the dimensions of the window.

```
truss_displacements_magnitude_viewport.offset(20,-10)
```

The **offset()** method is used on the viewport to offset the location of this viewport window from its current location by the specified X and Y coordinates. The offsets are floats specified in millimeters. This is done so that our windows are not one on top of another. It is not necessary to do this, it's only done here for aesthetic purposes and to demonstrate the **offset()** method to you.

```
truss_displacements_U1_viewport= session \
                     .Viewport(name='Truss Displacements at Nodes (U1 Component')
truss_displacements_U1_viewport.setValues(displayedObject=odb_object)
truss_displacements_U1_viewport.odbDisplay \
                             .setPrimaryVariable(variableLabel='U',
                                                 outputPosition=NODAL,
                                                 refinement=(COMPONENT, 'U1'))
truss_displacements_U1_viewport.odbDisplay.display \
                                        .setValues(plotState=(CONTOURS_ON_DEF, ))
```

```
truss_displacements_U1_viewport.setValues(width=250, height=160)
truss_displacements_U1_viewport.offset(40,-20)
```

These statements repeat the process except this time the SymbolicConstant is set to **COMPONENT** and the String to 'U1' in order to display the X component of the displacement. Also the window has been offset by a different amount in order to reveal the previous two underlying windows.

```
session.viewports['Viewport: 1'].sendToBack()
```

This statement uses the **sendToBack()** method to ensure that the default viewport window Viewport:1, which is the biggest window since we have not resized it, is behind all the newly created ones. In Abaqus 6.10 it is not really necessary since the newer windows automatically appear over the older ones but it might be helpful in older or newer versions of the software.

7.5 Summary

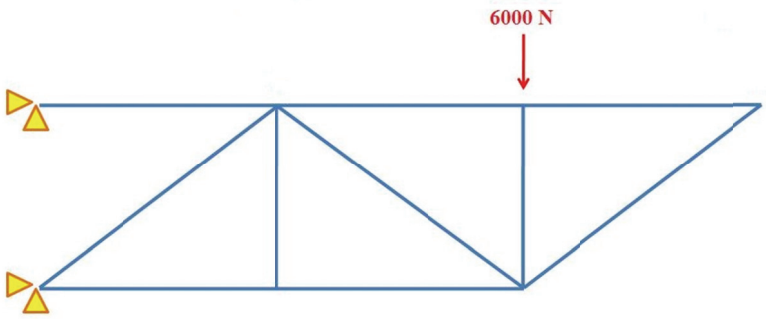
You just performed a 2D static truss analysis using a script. You are now familiar with the scripting commands most commonly used with such a simulation. Many of these commands will be used again in subsequent examples, just as ones from the Cantilever Beam example have been used here. There is no need to memorize these, you can always refer back to the examples in this book and copy and paste code suitably modifying it to fit your needs. Or you can use the replay file to assist you as well.

8

Explicit Analysis of a Dynamically Loaded Truss

8.1 Introduction

In this chapter we will perform a explicit analysis on a truss under dynamic loading. The problem is displayed in the figure. It is similar to the static general truss analysis of the previous chapter except that there is only one concentrated force and it is applied for 0.01 seconds.



In this exercise the following tasks will be demonstrated, first using the Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Identify sets

- Create a dynamic, explicit step
- Request history outputs
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job
- Retrieve history outputs

The new topics covered are:

- Model / Preprocessing
  - Create sets in the assembly
  - Change step time period and tell Abaqus to include non-linear geometry effects
  - Use history output requests specifying the domain and frequency of history outputs
  - Specify point of application of loads using sets
- Results / Post-processing
  - Plot history outputs
  - Save XY data of history output plots
  - Write XY data to a report
  - Display Field Output as color contours

8.4 Summary

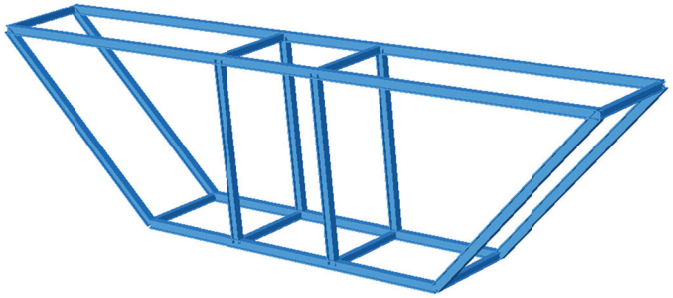
A few more concepts were covered in this chapter among which are creating sets, and post processing methods such as plotting XY data on a chart, and reporting it to an output file. We used some interesting tactics to discover the keys of the XY Data and latch onto it. These methods will likely be used by you in many scripts in the future.

9

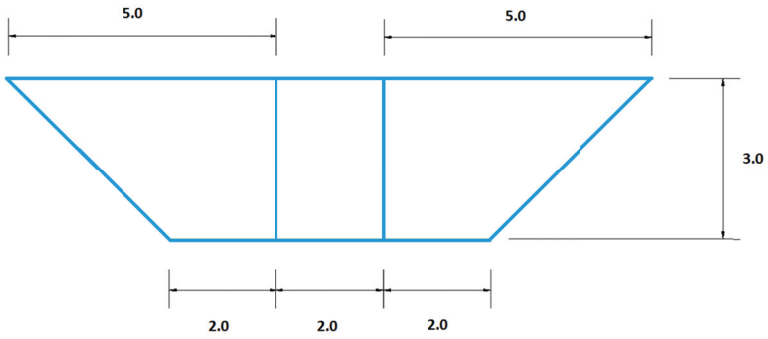
Analysis of a Frame of I-Beams

9.1 Introduction

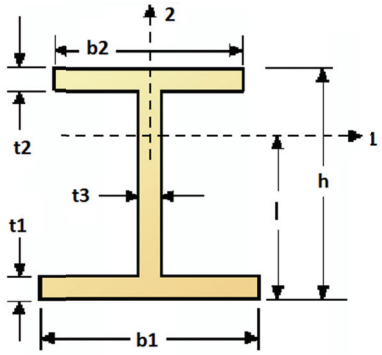
In this chapter we will perform an analysis on a frame made up of I-beams. The structure is displayed in the figure.



The dimensions of the beam frame are displayed in the following figure. All dimensions are in meters. In addition the distance between the two frames (ie, the length of the cross members) is 1.5 m.

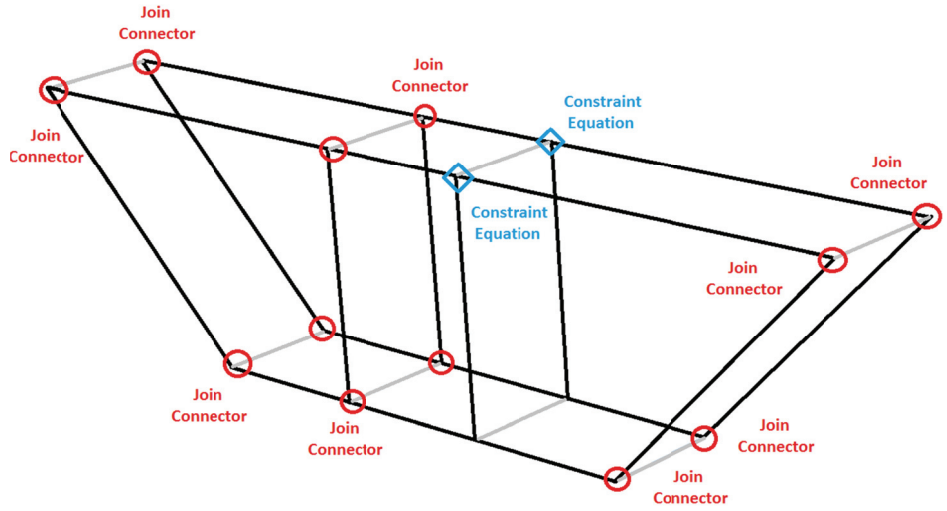


The beam profile dimensions are displayed in the figure.

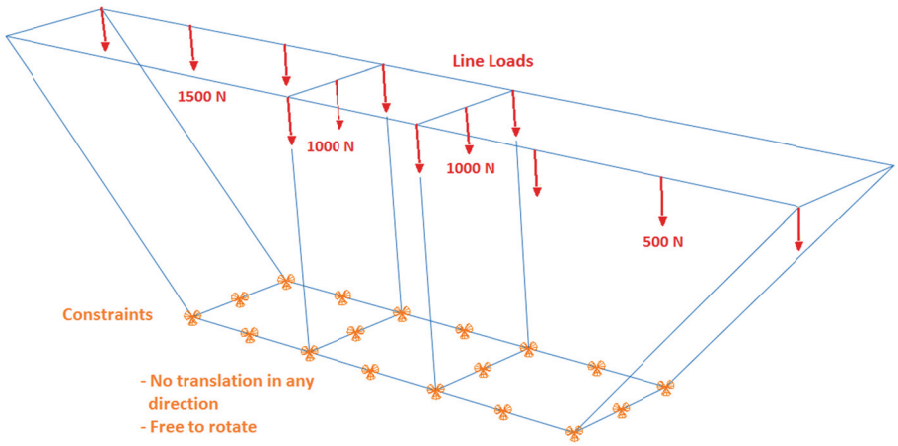


Dimension	Frame member	Crossbracing member
l:	0.075	0.06
h:	0.15	0.12
b1:	0.12	0.11
b2:	0.12	0.08
t1:	0.02	0.01
t2:	0.02	0.01
t3:	0.04	0.02

We will use both join connectors and constrain equations to create the pin joints between the frames and cross members in order to demonstrate how you can use both methods.



The loads and boundary conditions are displayed in the figure.



In this exercise the following tasks will be performed first using Abaqus/CAE, and then using a Python script.

- Create a part
- Create and offset datum points and datum planes
- Assign materials
- Create profiles
- Assign sections
- Set orientation
- Create an Assembly
- Create connector sections
- Perform connector assignments
- Identify sets
- Assign constraints with constraint equations
- Create a step
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job

The following new topics are covered in this example:

- Model / Preprocessing
  - Create a part starting with a reference point
  - Create datum planes and datum lines
  - Create beam elements in 3D using the ‘Create Lines: Connected’ and ‘Create Wire: Point to Point’ tools
  - Create beam sections and define beam profile geometry
  - Orient beams and render the orientations in the viewport
  - Use connectors (wire features + connector sections) to create joints
  - Use constraint equations to simulate joints
  - Use line loads

9.5 Summary

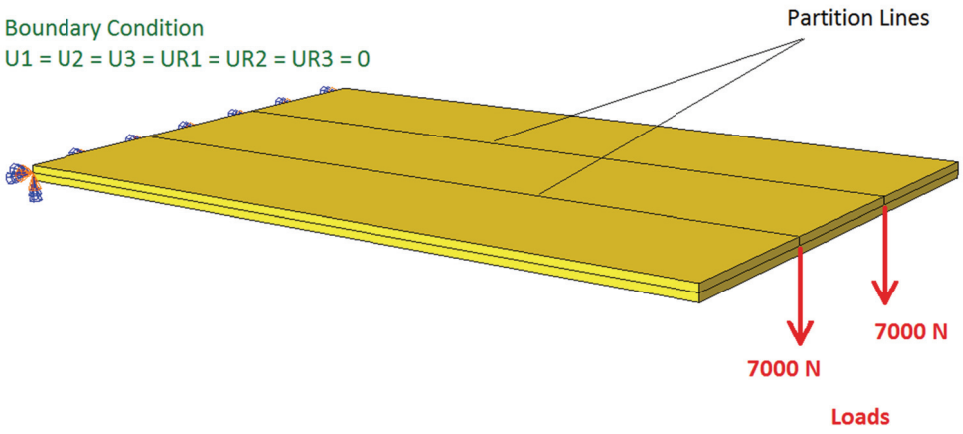
Some of the new topics covered in this chapter included creating datum planes and datum lines using a script. We also created connectors and constraint equations to simulate joints. You created a line load by using the **Region()** method a little differently to return a set-based region as opposed to a surface based one. These build on your knowledge of Python scripting in Abaqus.

10

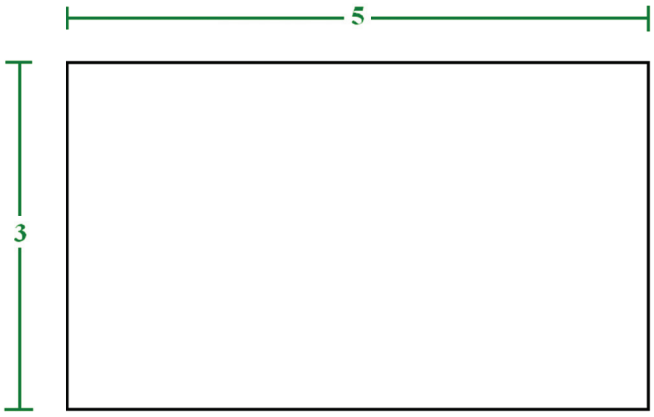
Bending of a Planar Shell (Plate)

10.1 Introduction

In this chapter we will perform a static analysis on a plate being bent by a concentrated force. The problem is displayed in the figure.



The dimensions are displayed in the following figure. All lengths are in meters and the shell thickness is 0.1 m.



In this example the following tasks will be demonstrated first using Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Create a static, general step
- Request field outputs
- Delete history outputs
- Create datum points and partition faces
- Assign loads
- Assign boundary conditions
- Create a mesh
- Create and submit a job
- Report field outputs to an external file

The following new topics are covered in this example:

- Model / Preprocessing
    - Work in 3D with a planar shell

  - Create sections of type ‘shell’, specify section integration properties and assign shell thickness
  - Define shell offset when assigning sections
  - Turn NLGEOM (non-linear geometry) option on/off as required
  - Delete history outputs
  - Create partitions for the purpose of generating selectable nodess
- Results / Post-processing
  - Show element labels on meshed model
  - Change the sort variable and sort order in the report profile
  - View/Change the work directory

(Remaining sections removed from preview)

10.5 Summary

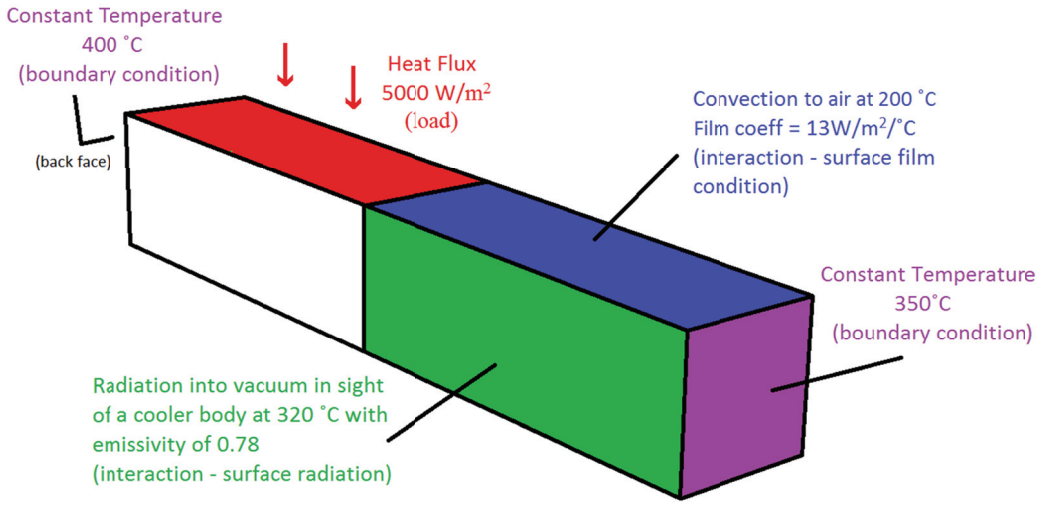
In this chapter we partitioned faces, displayed contours on a deformed plot, and reported field output to an external file. These are tasks you will undoubtedly script again in future.

11

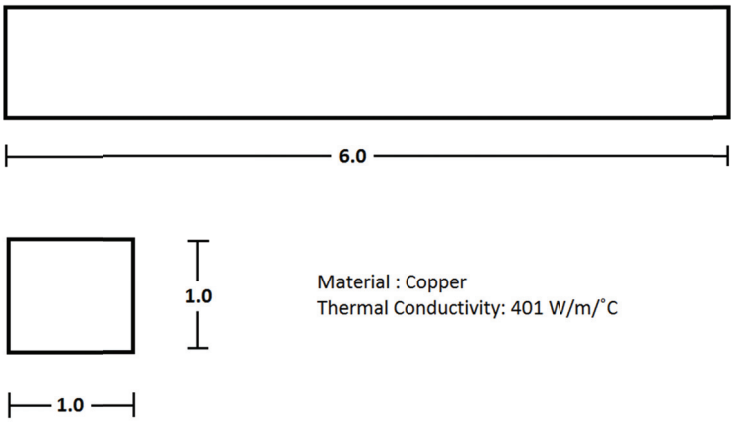
Heat Transfer Analysis

11.1 Introduction

In this chapter we will perform a heat transfer analysis on a rectangular block. The problem is displayed in the figure.



The dimensions and material properties are displayed in the following figure. The unit of length is meters.



In this exercise the following tasks will be performed first using the Abaqus GUI, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly
- Create a datum plane and partition a part
- Create a heat transfer step
- Assign boundary conditions
- Assign loads
- Create a mesh
- Create and submit a job
- Plot contours
- Change view orientation

The following new topics are covered in this example:

- Model / Preprocessing
  - Create a steady state or transient heat transfer step
  - Assign heat flux loads and constant temperature boundary conditions
  - Use interactions to define convection and radiation heat loss mechanisms

  - Modify model attributes to define the Stefan-Boltzmann constant and absolute zero of temperature scale
- Results / Post-processing
  - Display nodal temperatures as a color contour
  - Orient the viewport display and save custom views

(Remaining sections removed from preview)

11.5 Summary

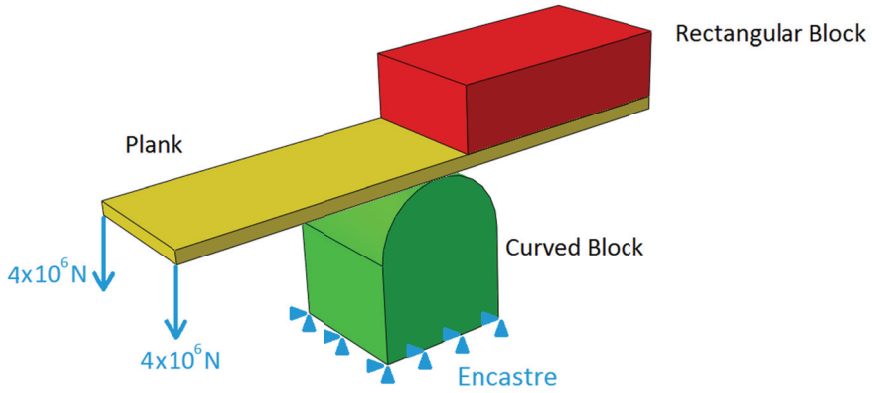
In this chapter we scripted a steady state heat transfer model. This included applying heat flux loads and constant temperature boundary conditions. You also learnt to change the primary variable in Abaqus/Viewer to plot a color contour and to change the camera angle. The heat transfer example used here was a very simple one, the aim was to introduce you to a few of the commands you are likely to use in a Python script. The Abaqus Scripting Reference explains in detail all of the options available to you for heat transfer analyses.

12

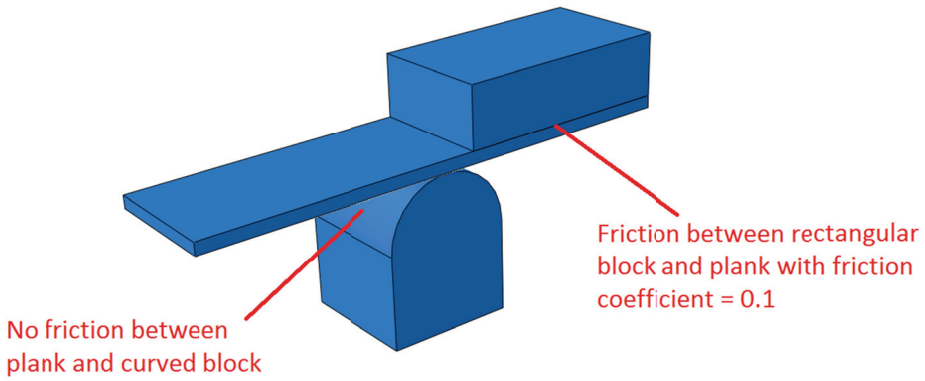
Contact Analysis (Contact Pairs Method)

12.1 Introduction

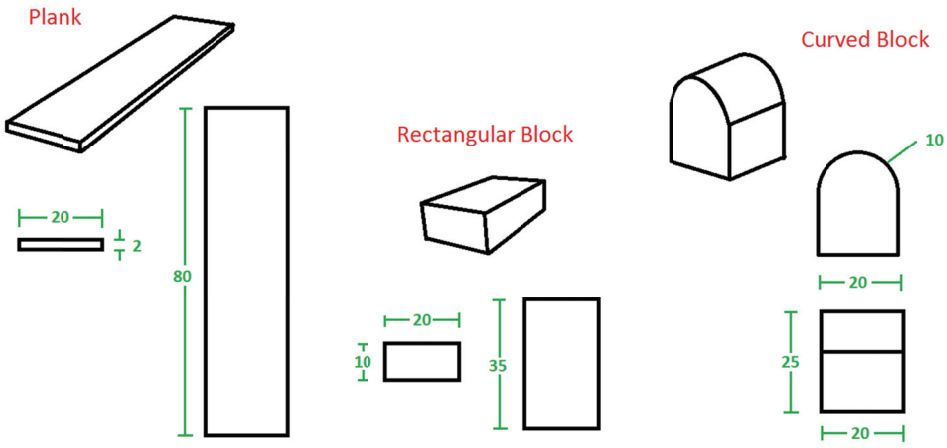
In this chapter we will perform a contact analysis. The problem is displayed in the figure. We will use the contact pairs method (as opposed to the general contact method).



We use frictional properties for the contact interaction between the rectangular block and the plank, and frictionless contact between the plank and the curved block.



The dimensions the parts are displayed in the figure. All dimensions are in SI with length in meters.



In this example the following tasks will be performed first using Abaqus/CAE, and then using a Python script.

- Create a part
- Assign materials
- Assign sections
- Create an Assembly using face to face constraints
- Create multiple steps
- Assign boundary conditions

- Assign loads
- Identify surfaces
- Assign interaction properties
- Create interactions
- Create a mesh
- Create and submit a job

The following new topics are covered in this example:

- Model / Preprocessing
  - Define surfaces in the assembly
  - Create interaction properties (specifically contact with and without friction)
  - Specify interaction pairs (contact surfaces)
- Results / Post-processing
  - Plot contact pressures to identify contact

(Remaining sections removed from preview)

12.5 Summary

In this chapter you worked with contact, created interactions and assigned interaction properties. Contact is commonly encountered both in real life and in simulations that you will be creating in Abaqus.

12.6 What's Next?

At this point we've worked through a number of model setups. Everything we've done so far could also have been implemented in Abaqus/CAE so you haven't really harnessed the power of scripting yet. In subsequent chapters we will reuse some of the scripts you have created here to demonstrate important concepts such as optimization and parameterization.

13

Optimization – Determine the Maximum Plate Bending Loads

13.1 Introduction

We've looked at a number of scripting examples over the last few chapters. In each of these examples we ran not just one aspect of a simulation, but rather the entire simulation from model setup to job execution to post processing using Python scripts. The benefit of having an entire simulation in the form of a script is that you now have the power to programmatically control it, parameterize it, add conditions and loops, and easily alter it for different scenarios. One of the primary uses of scripting is optimization.

In this chapter we shall look at an example of optimization using the planar shell (plate) bending model from Chapter 10. Let's assume you have a large supply of these plates and you'll be using them for construction or in a manufacturing project. It has been decided (for whatever reason) that you can save on material and component costs by maximizing the load borne by each plate. The materials expert has told you that the maximum allowable Mises stress in these plates is 35 MPa. You now need to figure out the maximum load these plates can withstand in bending while experiencing a stress less than 35 MPa in order to optimize your design. Since you aren't really modifying the plate based on the analysis, you aren't optimizing the design of the plate itself, however you will be optimizing your use of resources by loading each of the plates to their maximum capacity – and it is that maximum that you are tasked to find in this example.

13.2 Methodology

We wrote a script in Chapter 10 to run the plate bending simulation. We can modify this same script to run our optimization procedure. The majority of the script will remain the same. This includes the blocks that deal with model, part, material, section, assembly,

step, field output request, history output request (we didn't have any), boundary condition, partition and mesh creation. This means over 90% of the script remains unchanged.

The part of the script that needs modification is the application of the load. Since we are using the same concentrated forces and applying them at the same nodes, most of these statements will remain the same too. However we will put them inside a loop. At each iteration of the loop we will increase the magnitude of the concentrated forces. The block that creates and runs the job, as well as the post processing code, will need to be included inside of this loop so that the simulation can be rerun at each iteration of the loop and the results compared to our max stress criteria.

We will need to specify an initial force to use. We shall go with 5N. We will also need to specify how much to increase the force for the next iteration. We can go with a 5N increase at each iteration, so in the next iteration a 10N force will be applied, then 15N and so on. Each analysis job will be given a new name which states the amount of force applied such as PlateJob5N, PlateJob10N and so on. This way all the jobs will be listed in the model tree and output database list as they are created and run, and the user will be able to view the results of any of them if necessary. The results of each analysis will also be displayed in a new viewport which will pop-up over the previous one.

In the plate bending simulation a field output report file was written at the end. In this optimization we will continue to write this field output report file at every iteration. We will then read from this report, and extract the maximum stress from it. We will record this maximum stress by storing it in a file called 'iterative_analysis.txt' in a folder called 'Simulation results' so at the end of all the iterations we will have a table of force vs maximum stress. We will also compare this maximum stress to our maximum allowable stress of 35 MPa and if it has been exceeded we will break out of the loop.

At the end of the analysis we will highlight the elements of the plate which exceeded the maximum allowable stress and display the plate in the viewport so we can see at a glance where the stresses were too high. This gives me a chance to demonstrate how to change an element color within the visualization module.

(Remaining sections removed from preview)

13.5 Summary

After reading through this chapter you should now be able to perform an optimization by placing the bulk of your script inside of a loop and iterating through it. This is the standard procedure when performing optimizations using Python scripts. You also performed some of the most common file handling (input/output) tasks using the generated report files. In the process you were introduced to **try-catch** blocks for catching exceptions. And you learnt how to change the color of interesting elements in the viewport, adding to your knowledge of post-processing through a script.

14

Parameterization, Prompt Boxes and XY Plots

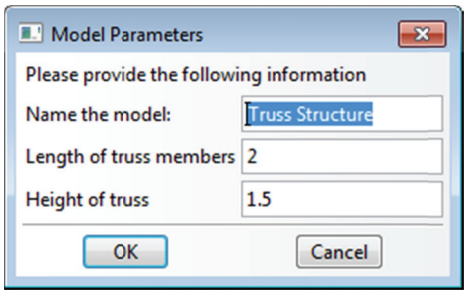
14.1 Introduction

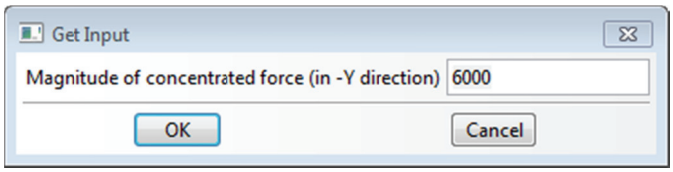
One of the most basic reasons for writing a script is that it gives you the ability to parameterize your model. This allows you to specify quantities in the form of variables whose values can be changed at runtime. If one of your dimensions is a variable, you can create your model geometry making use of that variable, and you'll then have the ability to change your model by changing that variable.

You already got a taste of this concept in the previous chapter with the plate, where the concentrated force was stored in the form of a variable whose value changed at every iteration. But this was a relatively simple example. You can in fact have many quantities in the form of variables which depend on the other variables. For example, you could specify the length of a truss member as a variable, and the cross sectional area as a variable which is related to the length by some mathematical relation. If you change the first variable, your script not only changes the length of the wire feature in the sketcher, it also changes the section properties accordingly. Or if you were working with beams you could have the script change the profile dimensions to make them some fraction of the length.

We will perform a similar parameterization in this chapter using the truss structure under dynamic loading from Chapter 9. In addition we will obtain the length of the beam members, as well as the magnitude of the concentrated force, as inputs from the user at runtime using prompt boxes. The ability to accept user input through a prompt box is a neat feature which allows the analyst to easily define a few variable values and observe

the response of the model. We will demonstrate the use of a prompt box which accepts one input, as well as a prompt box that accepts multiple inputs.





In addition we will revisit the XY plots created using history outputs, and play around with the plot characteristics. We'll change the characteristics and styles of the plot titles, axes, legends and so on. Quite often you will find yourself performing the same repetitive steps to visualize a result every time you run an analysis, and you can save some time and effort by writing these steps as a script. Although not the case in this example, it is quite popular to create standalone scripts for post-processing tasks which are only run after the analysis has completed.

14.2 Methodology

When the analyst runs the script, he or she will be prompted to type in the length of the truss members (they are all of equal length) and the height of the truss within a single prompt box. The script will be modified or parameterized so the part sketch will scale to these dimensions. The truss cross section area, which is a property assigned in the section module, will also be recalculated based on these dimensions. The radius of the truss members will be 0.05% of the length, and the cross section area will be calculated using this radius.

Recall that the **findAt()** method is used to find (and select) the truss members in order to assign section properties to them. Since the truss dimensions will now change based on user input, the locations of these members will also change, hence the arguments of the

findAt() method will need to be parameterized as well so they can dynamically update with the model geometry.

The user will also be prompted to enter the magnitude of the concentrated force, and this will be applied to the correct node (the one in the center). The history output will be requested from the node at the end of the structure. Note that the coordinates of both these nodes will depend on the geometry of the truss hence the **findAt()** method will once again be parameterized here.

(Remaining sections removed from preview)

14.5 Summary

In this chapter you saw a good demonstration of the parameterization procedure. Parameterization is the foundation of almost any optimization analysis as it allows you to treat quantities as variables and change them easily without having to recreate the model manually. In addition you now have a few blocks of script code that can modify all aspects of an XY plot, and you can reuse these in your own scripts.

15

Optimization of a Parameterized Sandwich Structure

15.1 Introduction

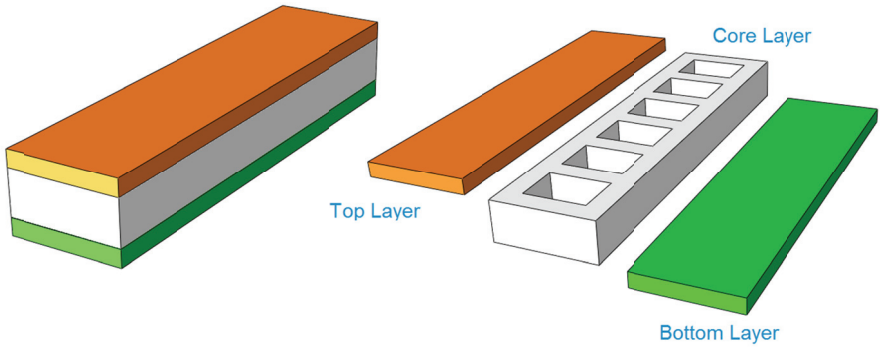
This chapter is another example of both parameterization and optimization studies. We will conduct an iterative optimization study on a parameterized sandwich structure. A sandwich structure consists of a layer of material sandwiched between two other layers which may or may not be of the same material. In our sandwich structure the two outer layers are solid planks or plates whereas the inner layer is a square honeycomb core. One end of the sandwich structure is fixed while the other end is free giving us something similar to a cantilever beam. Tie constraints will be used between the sandwich layers to hold them together.

We will write a parameterized script where the dimensions such as length, width, layer thicknesses and core cell dimensions will be specified at the start of the script, and the entire model will be created on the basis of these.

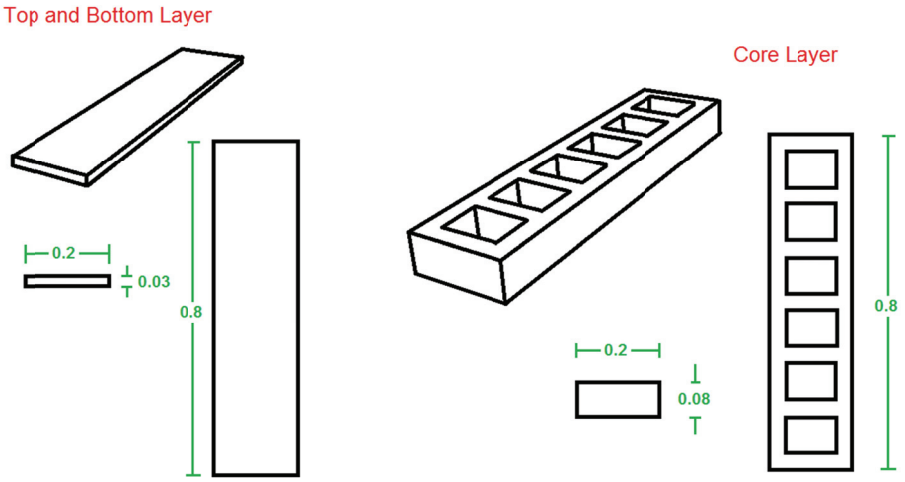
The user will provide input using a text file. Here each line of the text file will consist of tab separated values of all of the variables. For each line of this input file the script will extract the dimensions and perform an analysis. Therefore the bulk of the script will be inside a **for** loop iterating as many times as there are lines in the input file.

The results of each analysis (the displacement of nodes near the end of the sandwich beam) will be printed to an output file along with the input variables as tab separated values. The benefit of having such output is that these values can then be imported into a program such as Microsoft Excel or Matlab for creating plots and observing trends.

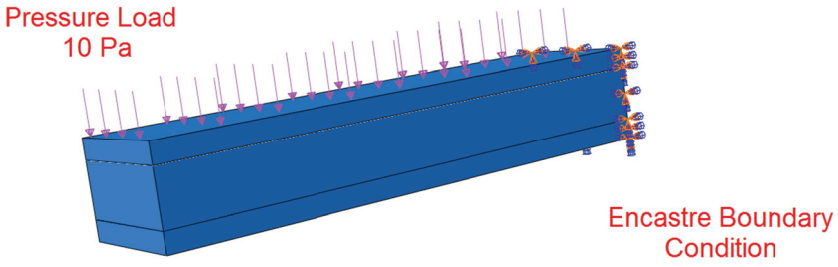
The geometry of our sandwich structure is displayed in the figure.



The following dimensions will be used:



The loads and boundary conditions are displayed in the next figure.



(Remaining sections removed from preview)

15.2 Summary

In this script you parameterized a complex model and ran an optimization on it. You read parameters from an input file, and spit out results into an output file. You now have a good idea of how parameterization and optimization are carried out using Python scripts. The output file can of course be imported into software such as Microsoft Excel or Matlab where the trends can be analyzed for optimization purposes.

16

Explore an Output Database

16.1 Introduction

This chapter is going to introduce you to reading output databases, and gaining useful information from them. When you run an analysis in Abaqus, the data you request – the field and history outputs – as well as other information, such as the geometry of the part instance, is written to the output database (.odb) file. You might be required to extract some specific information from an odb as part of your analysis procedure. A script might be a more efficient then manually using the Abaqus/Viewer environment. In addition there are some tasks that are impossible to perform in the Viewer but possible through a script.

In this example we will experiment with the output database of the static truss analysis from Chapter 7 and the explicit dynamic truss analysis of Chapter 8. We will perform 4 tasks.

1) We will extract the stress field, and display a contour plot of one-half of its value. Each of the truss members will therefore appear to have only half of their original stress when viewed in Abaqus/Viewer. While this may not appear very useful, the purpose is to demonstrate how you can modify a field by performing a mathematical operation on it or a linear combination with another field. We will use the field output data of the static truss analysis for this.

2) We will extract information about the part instance used in the analysis, its nodes and elements, and find out which element and node experienced the maximum stress and displacement respectively. You saw an example of finding which element experiences the maximum stress in the plate optimization example (Chapter 13), but in that example you obtained this information by reading the

report file generated during post-processing. This time you will read the output database. You will also use the print command in a manner similar to the printf() command from C which allows you to format your printed output. We will use the field output data of the static truss analysis for this.

3) We will find out what regions of the part have history outputs available, what these history outputs are, and extract the history output data. You will also see how to find out which sets were defined in the model, and how to extract information about the history region these sets correspond to. History output information will be examined for both the output databases – the static truss analysis and the dynamic explicit truss analysis.

4) We will extract the material and section properties from the odb. We will also extract the entire material and section definitions from the static truss analysis odb and put them in a new Abaqus/CAE model for future use using some built-in methods provided by Abaqus.

In the process you will also learn of the various type of print statements, and how to format printed output to suit your needs (and also to make your code more readable). In addition you will discover the **hasattr()** and **type()** built-in functions offered by Python.

Performing these tasks will give you a good insight into working with Abaqus output databases using a Python script.

16.2 Methodology

For the first task, we will read in the stress [S] and displacement [U], both **FieldOutput** objects. We will divide the stresses by 2 to make them half their value, and leave the displacements at their present values. We will then create a new viewport window, set the primary variable to our new half stresses, and the deformed variable to the unchanged displacement, and plot these. We will also turn on element and node labels, so we can see the element and node numbers in the viewport to better understand what is going on in the next task.

For the second task, we will use the object model to examine field output values in the output database. Output databases consist of a very large amount of information, and this information is buried inside the object model at different levels –you have containers with information and more containers nested within them with additional information. To

find the element with the maximum stress and the node with the maximum displacement, we will need to loop through all the elements and nodes examining their stress and displacement values respectively.

For the third task we will once again use the object model, but this time we will examine history output information.

For the fourth task we will use some methods provided by Abaqus to easily extract material and section information from an odb. We will create a new model file and place this information in it for demonstration purposes.

16.3 Before we begin – Odb Object Model

(Section removed from preview)

16.4 How to run the script

Open a new model in Abaqus/CAE and run the script created for the static truss analysis using **File > Run Script…** The analysis will create an output database file 'TrussAnalysisJob.odb' and the script will open and display it in the Abaqus/Viewer viewport.

Then then open another new model in Abaqus/CAE and run the script created for the dynamic explicit truss analysis using **File > Run Script…** (It will be necessary to open a model to run the second script since both the scripts were originally written to be standalone and assume the existence of a default model 'Model-1' which they rename). The analysis will create an output database file 'TrussExplicitAnalysisJob.odb' and the script will open and display it in the Abaqus/Viewer viewport.

The reason both these scripts must be run is that they run the analysis and produce the output databases. The Odb exploration script in this example needs to access these output database files.

Once these scripts have been run, the Odb exploration script written in this chapter can be run using **File > Run Script..** either with those models still open in Abaqus/CAE, or in a

new Abaqus/CAE model. (It does not make a difference since this script only accesses the .odb files and does not assume the existence or lack of any model in Abaqus/CAE).

(Remaining sections removed from preview)

.

16.5 Summary

You now have a good understanding of how you can access information stored in an output database using a Python script. There is a wealth of information available in an odb, and all you need to access it is a basic understanding of the output database object model. There is no sense in memorizing the entire tree structure which has hundreds of nested repositories, attributes and methods; you should instead use object model interrogation with **print** and **prettyPrint()** statements to determine how to access the information you need.

17

Combine Frames of two Output Databases and Create an Animation

17.1 Introduction

In the previous chapter we explored two output databases to understand the output database object model and learn how to obtain useful information from an .odb file. In this chapter we will demonstrate how to create a new output database file from scratch. To make things interesting we will open two other output databases, extract the required information from them, and combine this information from both of them into a new output database.

We will modify the plate bending example from Chapter 10 in order to include the effect of plasticity, and increase the loading on it to force it into plastic deformation. We shall request Abaqus to write restart information to the .res file during this analysis. We will then continue the analysis using the restart file and remove the load from the plate allowing it to spring back and recover its elastic deformation (the plastic deformation will not be recovered). The two analyses will generate two output databases. However these do not overlap, and the first frame of the restart analysis will correspond to the last frame of the original analysis. In order to view the results of the original analysis in Abaqus/Viewer, the first .odb needs to be opened, and for the second analysis (springback) the second .odb will need to be opened.

Our goal is to use a Python script to read both the output databases, extract the nodal displacement information, and create a new output database which combines the information of both analyses. This allows the analyst to view the entire set of results (that you choose to include in the combined odb) in Abaqus/Viewer since the frames of both

analyses are joined together. We will then create an animation which includes both the bending and the springback.

17.2 Methodology

We will need to create 3 Python scripts for this example.

The first script will be a modification of the plate bending script from Chapter 10. We will update it to include plastic material properties, and increase the load to cause bending stresses that exceed the elastic limit. We will also need to request Abaqus to write restart information to the .res file. On running the simulation an output database file will be produced.

The second script will replicate the original model, and add a new step to it where the load is removed. It will then continue the analysis using this new model. On running this simulation a second output database file will be produced.

The third script will open and read the output databases created by the two analyses, and extract the nodal displacement information. It will then create a new output database, and in it create the part, instance it, create two steps, and add the displacement field output data to these steps from each of the .odb files. It will then open this .odb in Abaqus/Viewer, animate the time history and save the animation, which will include both the bending and the springback.

17.3 Summary

In this chapter we extracted data from 2 existing output databases and created a new one using this information. You now have a firm understanding of not only how to extract information from output databases using a Python script, but also how to construct one from scratch. Using this technique you can create output databases that contain only what you need - either for further processing tasks or to help you or another analyst visualize specific results.

18

Monitor an Analysis Job and Send an Email when Complete

18.1 Introduction

A single analysis job in Abaqus can take hours or even days to run. Multiple jobs running as part of an optimization routing can take a considerable amount of time to execute. It is possible to write a script that monitors a job and provide updates to the analyst.

In this example we shall monitor the running of the Cantilever Beam example from Chapter 4. We shall detect when the job completes or aborts. We will then log into a Gmail account, and send an email to another address informing the analyst that the job has either completed running or quit with errors.

18.2 Methodology

In our original Cantilever Beam script we submit the job and then wait for it to complete using the **WaitForCompletion()** function. On completion, program control returns to the script and subsequent statements, in our case post processing statements, are executed.

We will no longer use the **waitForCompletion()** function. Instead we will use the **addMessageCallback()** function of the **MonitorMgr** object provided by Abaqus to monitor messages generated by Abaqus during the analysis. Every time a message is generated a function **jobMonitorCallback()**,defined by us, will be called, which will check the type of the message. If the message type is either **ABORTED** or **COMPLETED** it will call another function **postProcess()**, also defined by us, to log into Gmail's SMTP server and send an email indicating that the job has been completed (or aborted).

(Remaining sections removed from preview)

18.3 Summary

In this chapter you were introduced to job monitoring. In the example script we monitored the messages **ABORTED**, **ERROR** and **JOB_COMPLETED**, which are only a few of the available message types. If job monitoring is an important topic in your work I strongly recommend looking up the other message types and experimenting with them. We also learnt how to send an email from a Python script. While this involved some advanced Python programming, it not only gave you some reusable code in case you wish to have your jobs email you on completion, but it also demonstrated the fact that you can harness powerful features of the Python language and are not only limited to Abaqus kernel commands.

PART 3 – GUI SCRIPTS

Up until this point all the scripts you have written have run without much interaction with the analyst, with the exception of the prompt boxes of Chapter 14. This is perfectly acceptable for most scripts, and possibly all scripts you ever write for Abaqus will be like this. However there may be times when you wish to create an interface for your script, just so you can type in values or select options at runtime. If you work in an environment where other analysts will be using your scripts, a visual interface can save them having to modify your scripts directly, and may therefore be beneficial for everyone involved. Taking things a step further, if you are in a large organization where individuals without much Abaqus experience will be working with your models, you may wish to alter the Abaqus/CAE interface itself so as to provide them with a pre-determined workflow and limit their exposure to the complexities of Abaqus.

In Part 3, you will learn how to create simple dialog boxes using the Really Simple GUI (RSG), as well as custom interfaces and vertical applications using the Abaqus GUI Toolkit. From my personal experience, most individuals working with Python scripts in Abaqus are not required to create GUIs, therefore most of the following chapters can be considered optional for most readers. However it wouldn't hurt to skim over them, just so you get an idea of what is involved.

The last chapter of the book deals with Plug-ins. These are useful for both kernel and GUI scripts, so browse through it even if you skip chapters 19 – 21.

19

A Really Simple GUI (RSG) for the Sandwich Structure Study

19.1 Introduction

In Chapter 15 we wrote a parameterized script to study the deflection of a pressure loaded sandwich structure. This script accepted parameters using a specially formatted input file and ran a complete analysis for each set of inputs. In this chapter we shall modify that script to instead accept inputs/parameters using a dialog box presented to the analyst in Abaqus/CAE. To simplify the example and focus on topic at hand, the analysis will only accept one set of inputs and run once using these. The dialog box will only be presented once at the beginning and there will be no looping.

The dialog box will be created using a facility known as the Really Simple GUI, abbreviated as RSG. RSG allows the analyst to quickly create a dialog box with text fields, checkboxes, combo boxes (dropdown menus), radio buttons and so on without using any complex GUI customization tools. The drawback is that you can only customize the appearance of the dialog box you create, not the rest of the Abaqus/CAE interface. In addition, the appearance of the dialog box itself cannot change dynamically, meaning that you cannot show and hide controls, or display different options based on previously selected ones.

19.2 Methodology

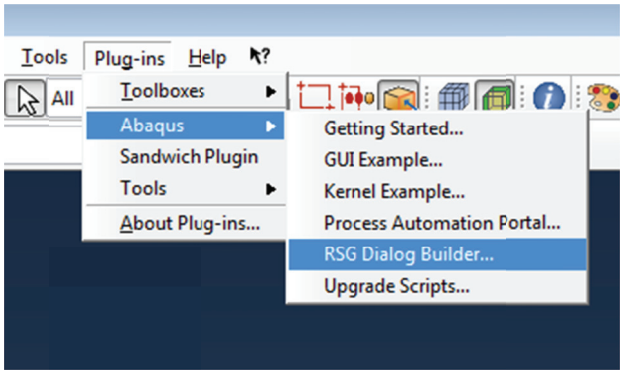
We will modify the script from the sandwich structure analysis. It will be placed inside a function using the **def** keyword. This function will be called by the RSG dialog box when the user clicks OK, and the parameters provided to the script will be the values supplied by the user using the dialog box controls. Needless to say we will delete the parts of the

script that read data from an input file. In addition the loop itself will be removed since the analysis will only be run once.

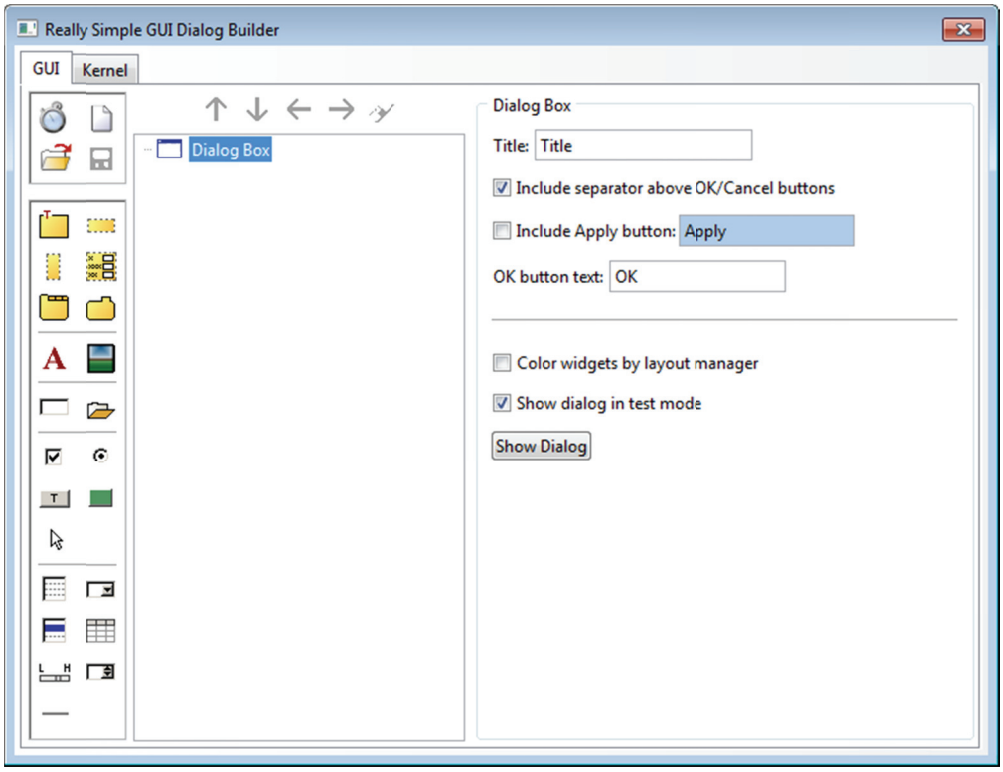
The RSG Dialog builder will be used to create the dialog box. It is a WYSIWYG (what you see is what you get) interface where you select which controls you would like to place on the dialog box from the available options, and the finished product will look identical to it.

19.3 Getting Started with RSG

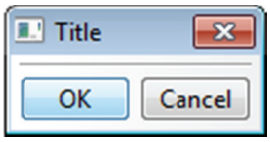
In Abaqus v6.10 the RSG Dialog builder can be accessed from **Plugins > Abaqus > RSG Dialog Builder…** as displayed in the figure.



The Really Simple GUI Dialog Builder appears as shown in the following figure. On the left hand side you see a set of tools you can use. Most of these are controls/widgets that can be added to the dialog box. As you click on them they will populate the tree in the center giving you a hierarchy which can be rearranged using the arrow keys.



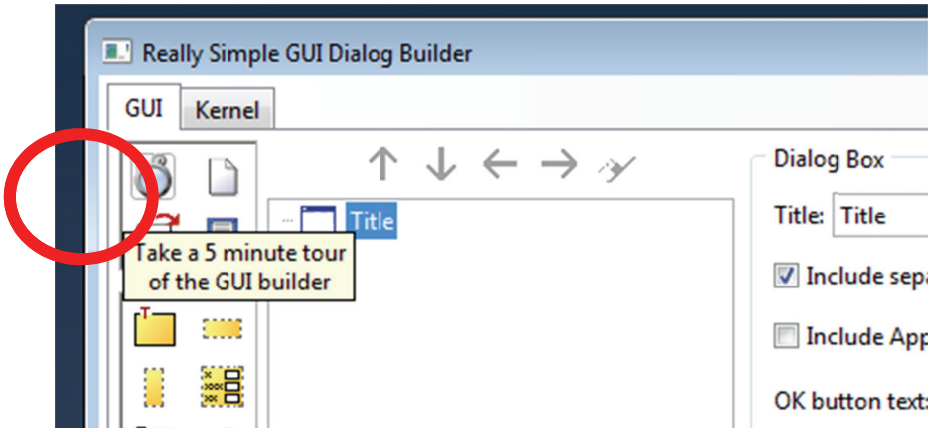
In the right side of the window, where you see a few dialog box options, check 'Show dialog in test mode' and click the 'Show Dialog' button.



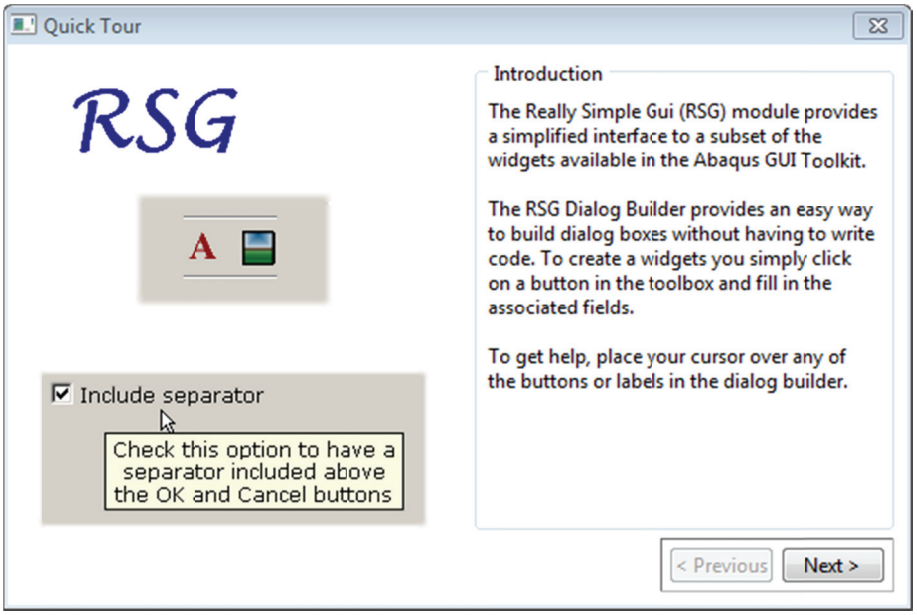
A dialog box is displayed. At the moment you haven't added any controls to it hence all it contains is **OK** and **Cancel** buttons.

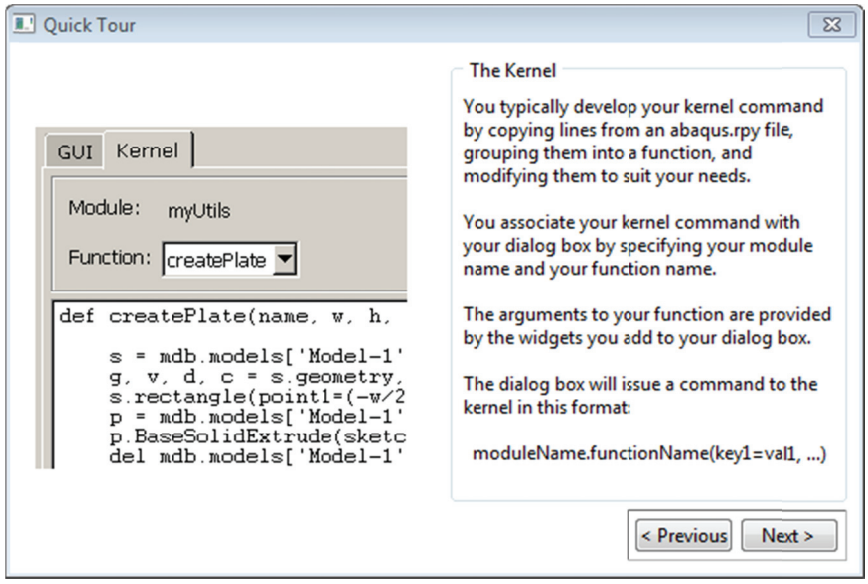
The RSG comes with a basic 5 minute (or shorter) tutorial. It makes little sense for me to rehash what is already covered in this tutorial especially since it is available to everyone. You can either run through it in Abaqus, or follow along using the screenshots below. These screenshots were taken in Abaqus/CAE Student Edition 6.10-2.

Click on the "Take a 5 minute tour of the GUI builder" tool.

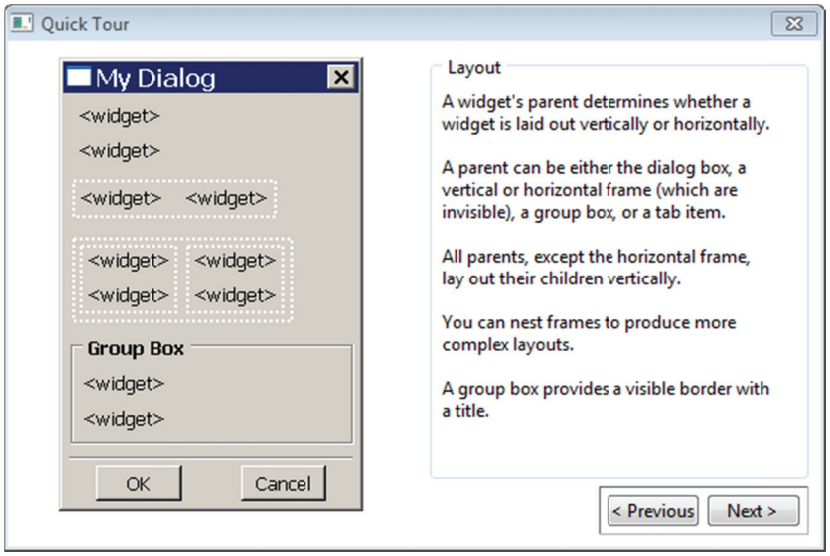


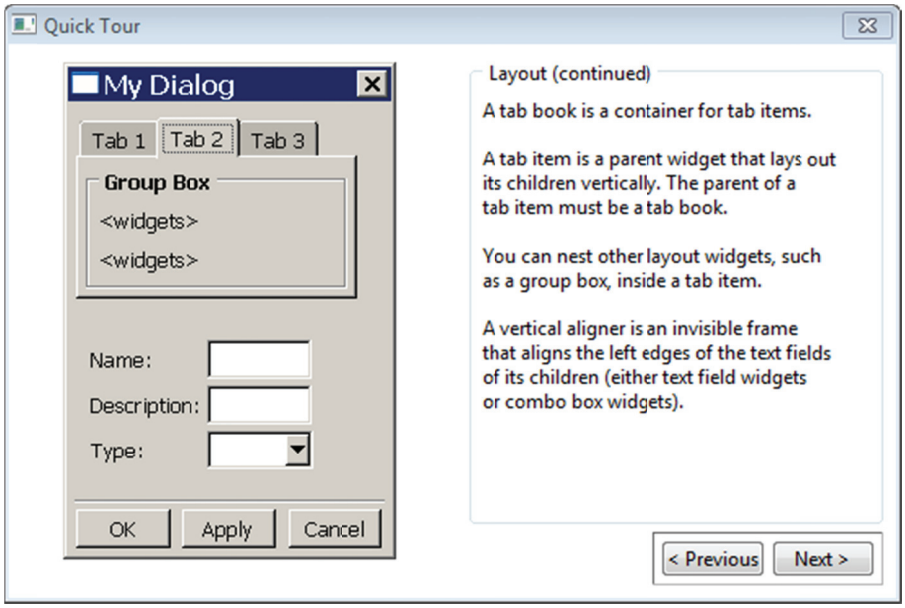
The ‘Quick Tour’ begins.

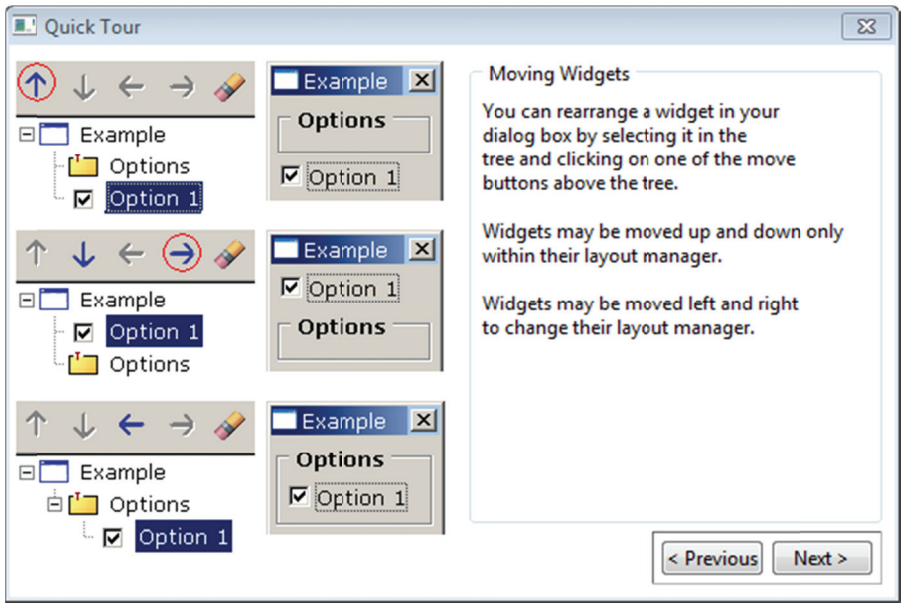




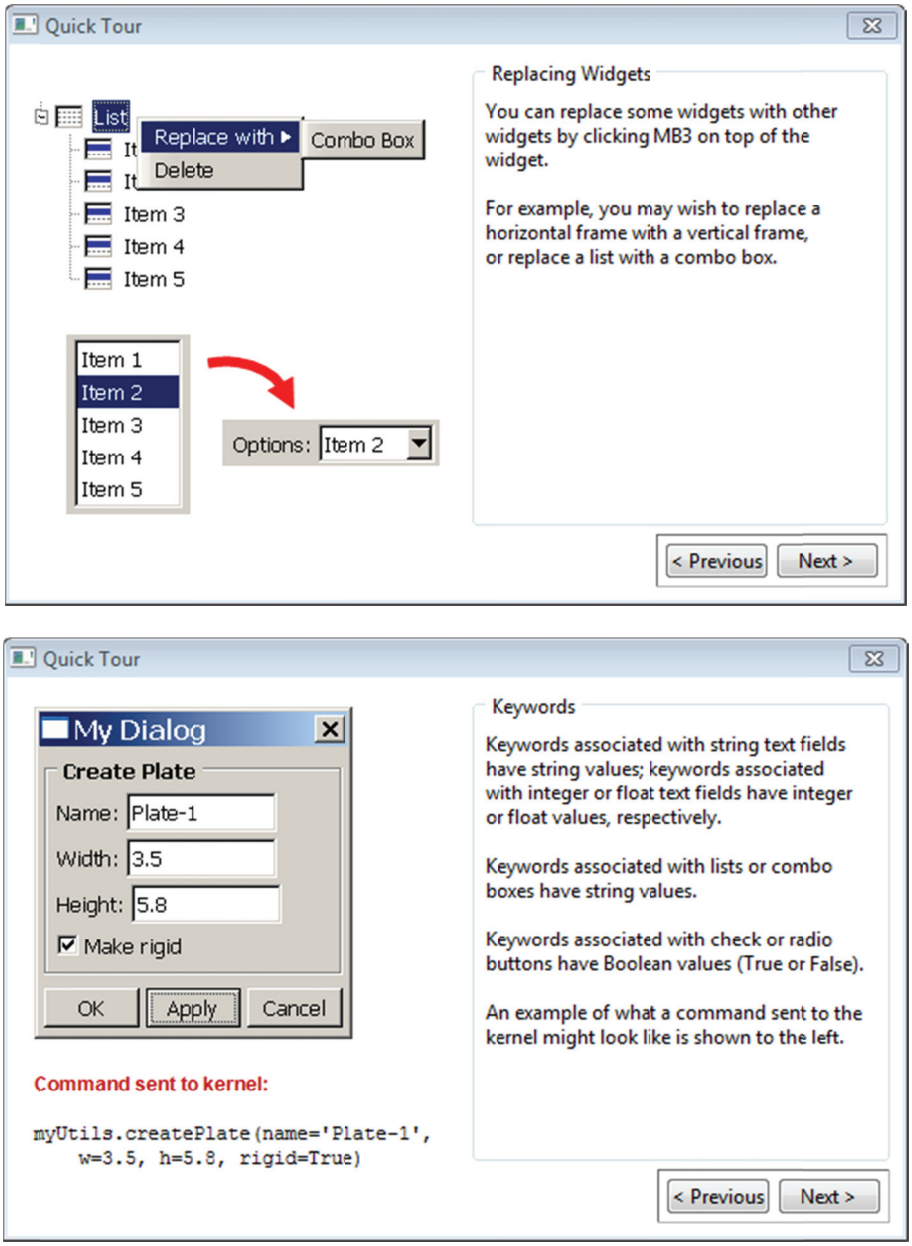
This window is where we will link the RSG to our Python script. The script itself will form what is labeled at the module, and the function within the script will be the function called when the **OK** button is clicked in the dialog box. In the above figure, the module is 'myUtils' and the function is 'createPlate', which means that a function called 'createPlate()' will be called in a script called 'myUtils.py'.





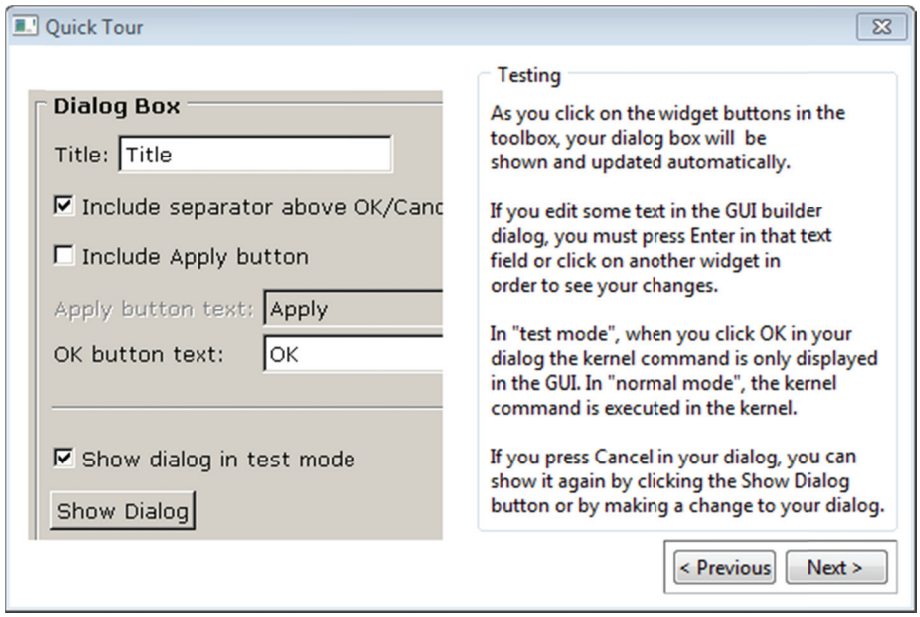


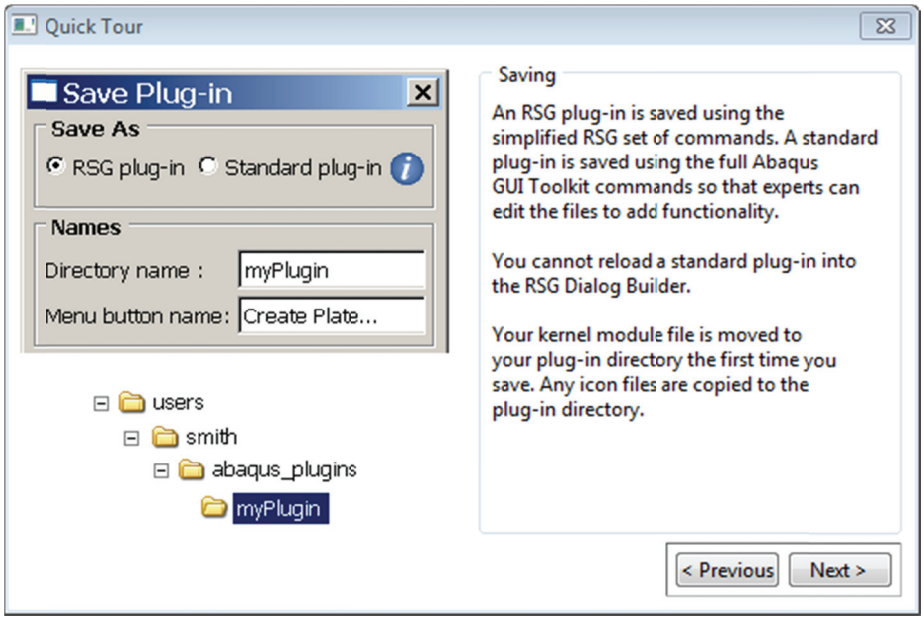
Moving widgets up and down tends to change their position in the dialog box. Moving widgets left and right allows you to nest them within a layout manager thus allowing them to be affected by the layout.

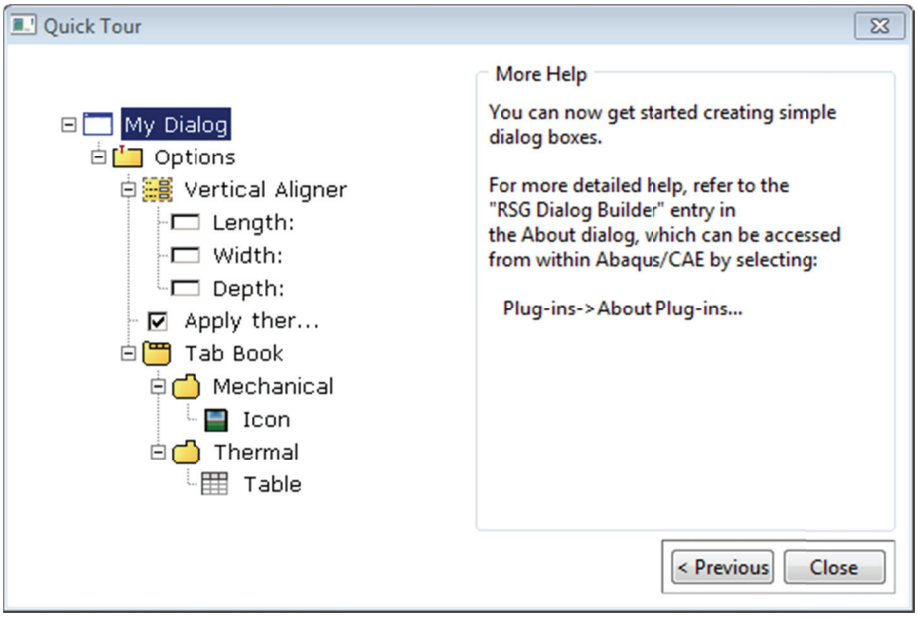


You associate keywords with each widget of the dialog box and also define the type of data it accepts. Here the text fields for name is given the keyword ‘name’ and accepts Strings. The other two fields are assigned the keywords ‘w’ and ‘h’ and accept floats. The

checkbox's keyword is 'rigid' and it always returns a Boolean. These keywords and their values are passed to the function associated with the dialog box as parameters.



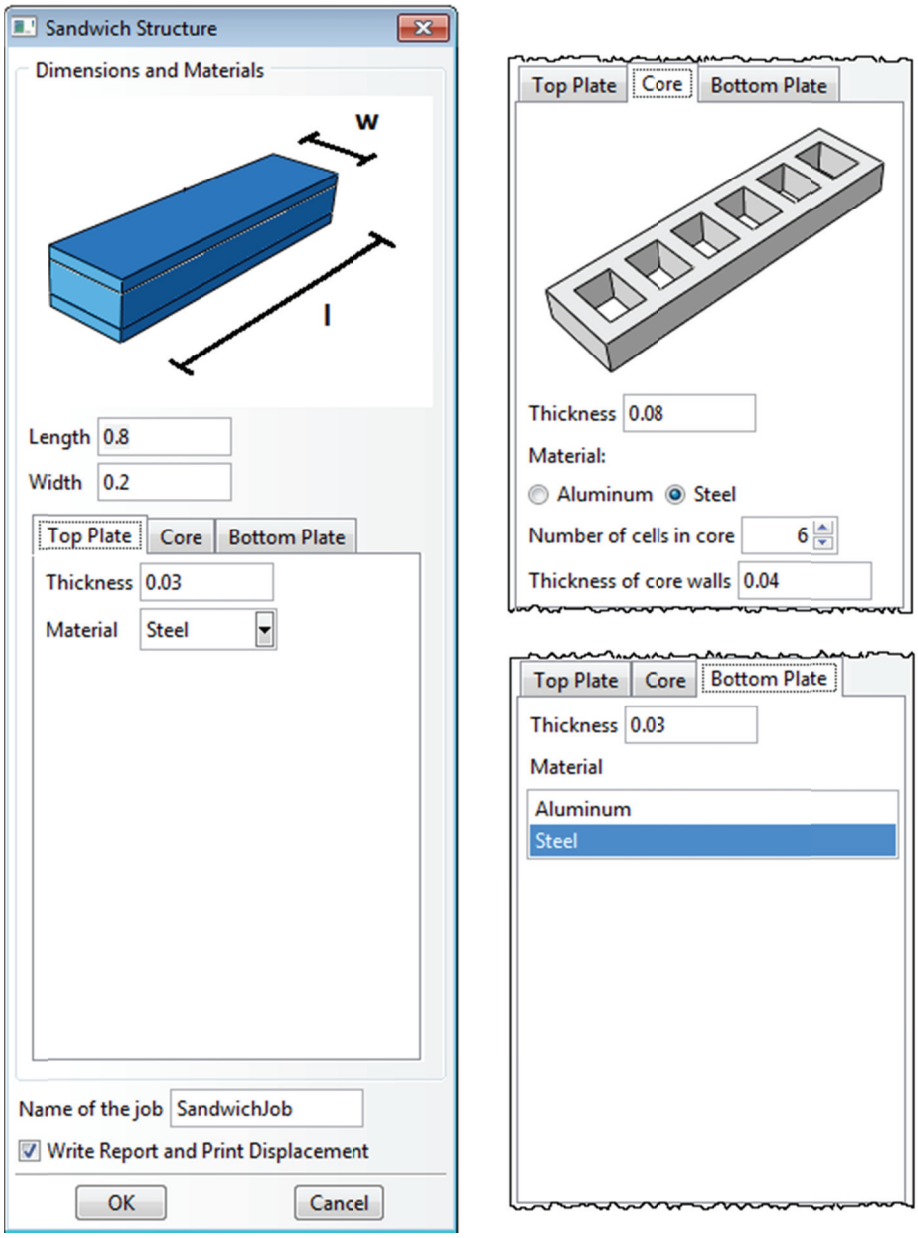




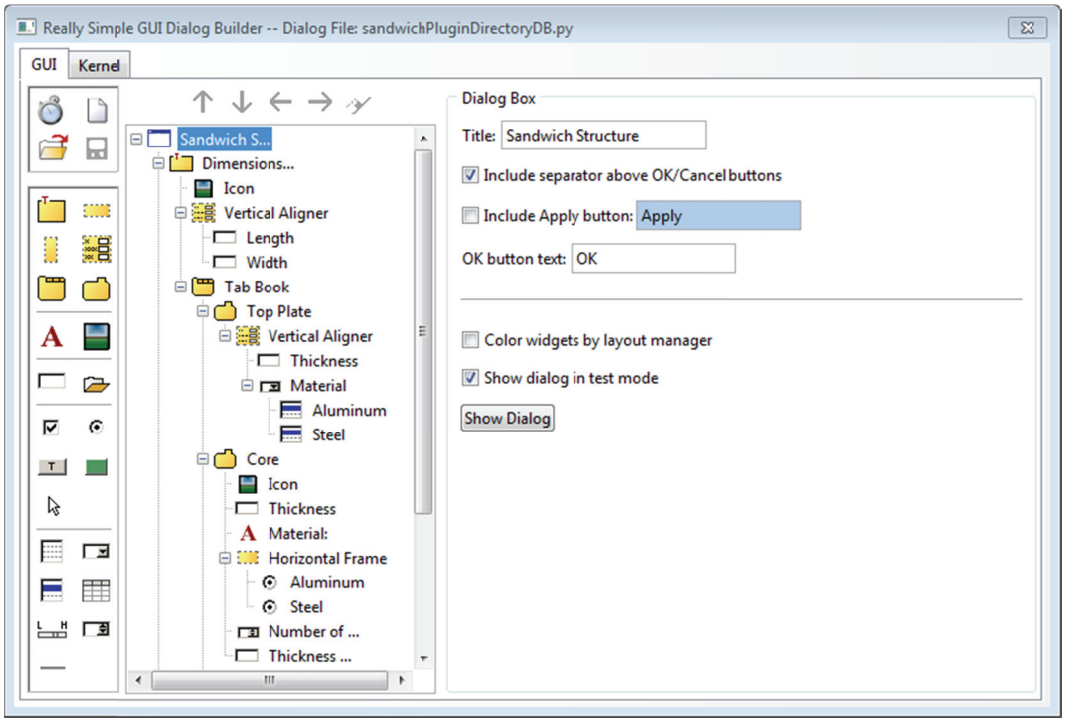
19.4 Create an RSG for Sandwich Structure Analysis

Now that you've run through the 5 minute tutorial and got an idea of how RSG works, let's work through our example. I have already gone ahead and created a GUI dialog box. Laying the widgets out onto the canvas is simple enough but you should try it once and obtain the same layout that I have here.

Here is what our RSG dialog box will look like:

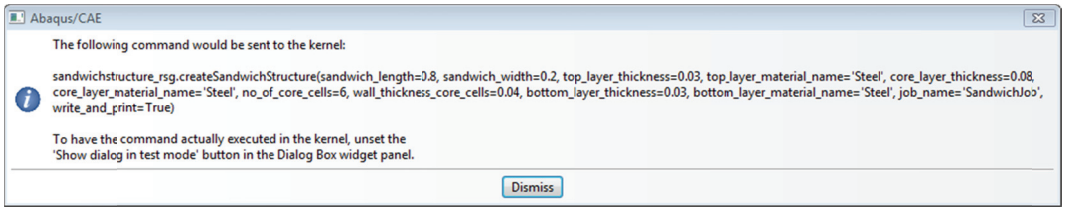


Lets focus on the parameters used to create this.



Here you see the settings for the plugin. The title 'Sandwich Structure' will appear in the title bar of the dialog box. We are including a separator, which is a horizontal bar, above the OK and Cancel buttons by checking the option. We have set the OK button text to the default of "OK" although you can change it to something else if you prefer.

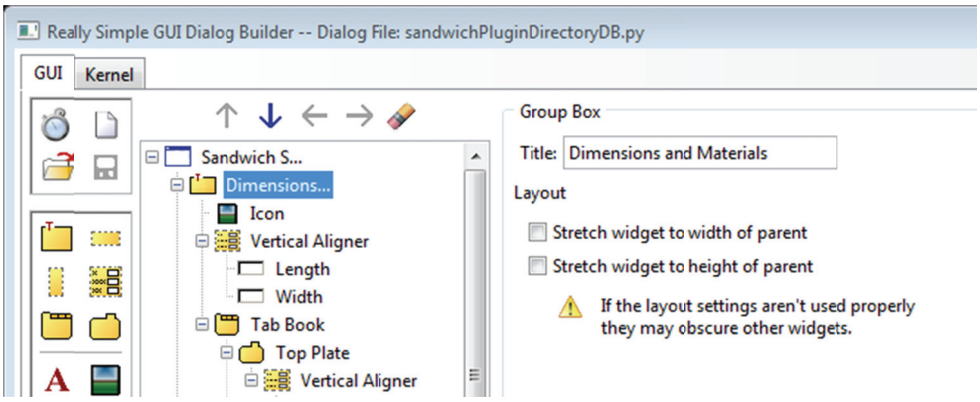
If you click the 'Show Dialog' button, you will see the dialog box. 'Show dialog in test mode' is currently checked for testing purposes. This means that when you click OK Abaqus will not actually run the script. Instead it will display a message:



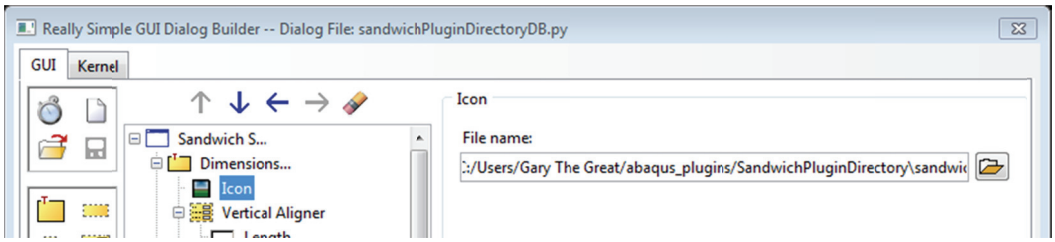
Abaqus indicates that it will call the **createSandwichStructure()** method in the Sandwichstructure_rsg.py file with the statement

```
Sandwichstructure_rsg.createSandwichStructure(sandwich_length=0.8,
sandwich_width=0.2, width=0.2, top_layer_thicker=0.03,
top_layer_material_name='Steel', core_layer_thickness=0.08,
core_layer_material_name='Steel', no_of_core_cells=6, wall_thickness_core_cell=0.04,
bottom_layer_thickness=0.03, bottom_layer_material_name='Steel',
job_name='SandwichJob', write_and_print=True).
```

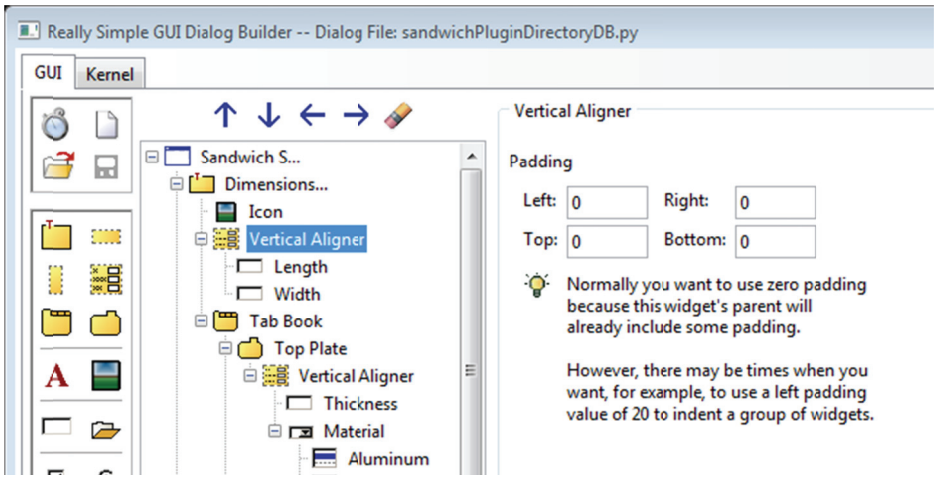
All the widgets are placed inside a group box which we have given the title 'Dimensions and Materials'.

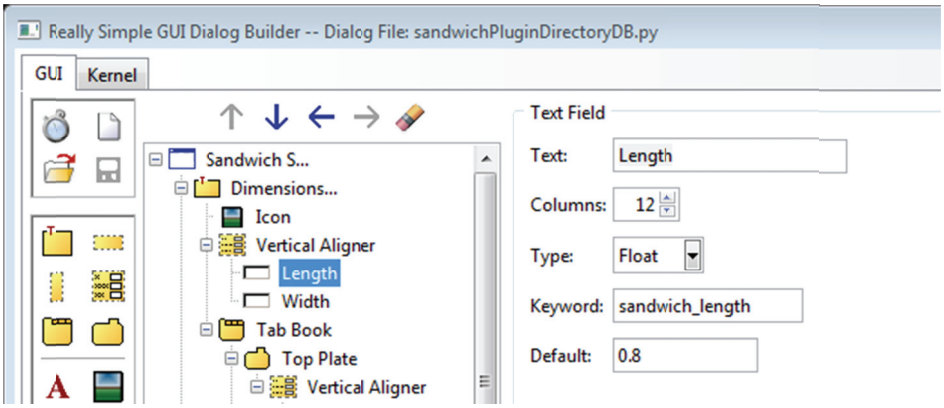


An icon widget is used to add the image. The path to the image is specified here.



We create a vertical aligner widget to position the length and width text fields vertically. Any items placed inside a vertical aligner are automatically positioned vertically. We will not apply any padding to this vertical aligner.

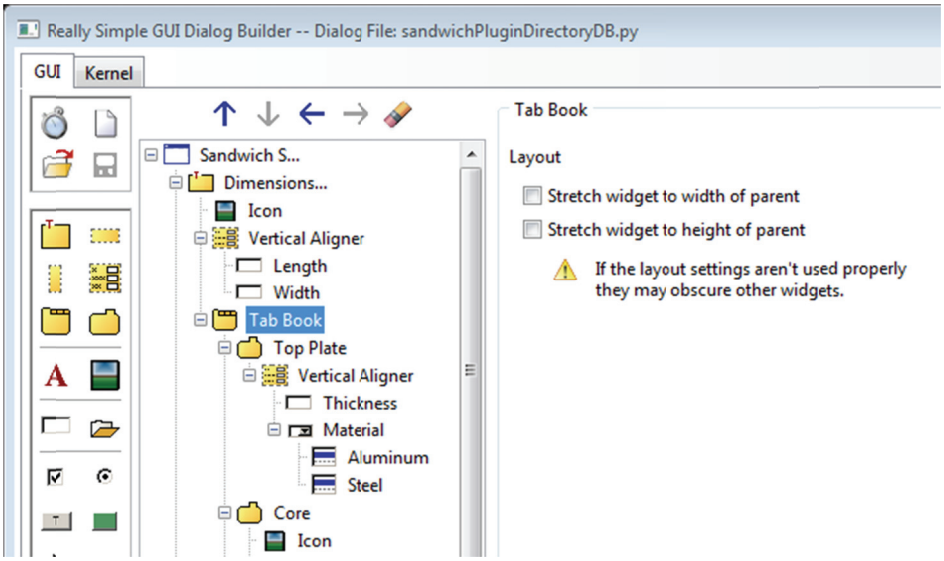




The length text field is defined here. The text is set to 'Length' hence the word 'Length' will appear next to the text field on the canvas. The number of columns is set to 12 meaning that 12 characters will be visible in the text field. You can actually type more characters, but the whole line will shift left as you type more and you will only be able to see 12 characters/digits. This is more than enough room for our purposes. The type is set to 'Float' indicating that a float value is expected here and this will be passed to a float

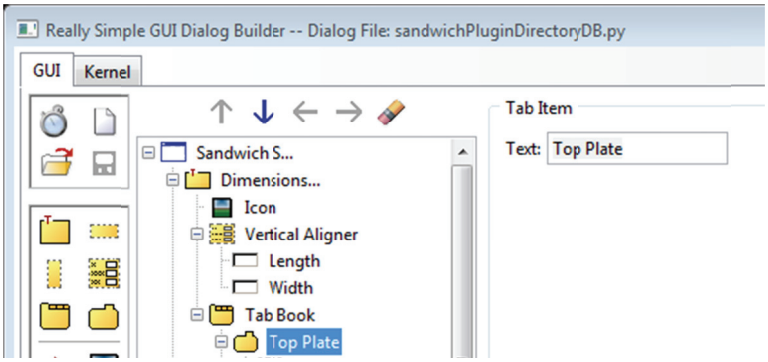
variable. The keyword **sandwich_length** is associated with this text field, hence when the **OK** button of the dialog box is pressed the function **createSandwichStructure()** will be passed the parameter **sandwich_length=xyz** where **xyz** is the float entered by the user. The default is set to 0.8.

The definition of the width text field is similar. It is assigned the text 'Width', the keyword associated with it is **sandwich_width** and the default value is 0.2.

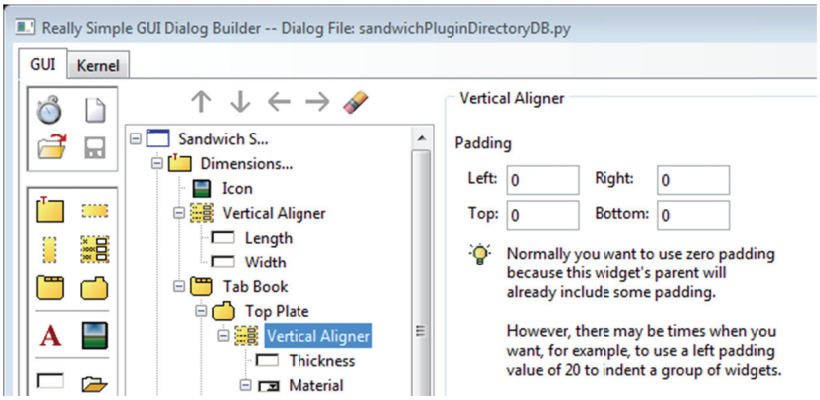


A tab book widget is used to create a tabbed section. Each of the tabs – Top Plate, Core and Bottom Plate will be individual containers nested within the tab book container.

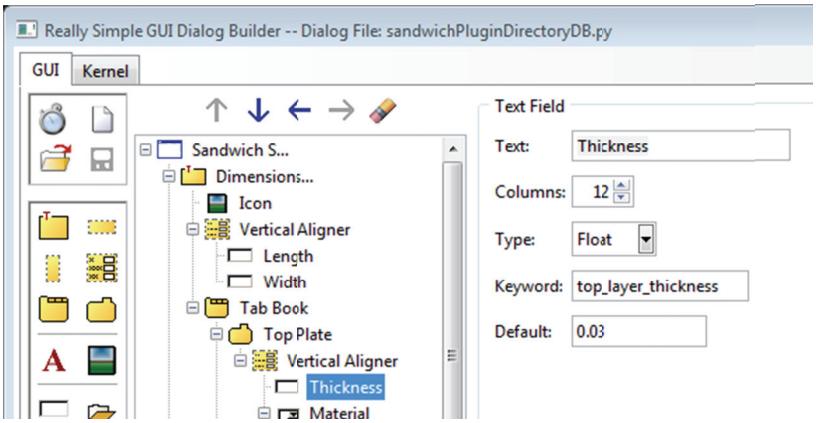
The Top Plate container will accept settings for the top plate. We give it the title 'Top Plate' which appears as the name of the tab in the tab book.

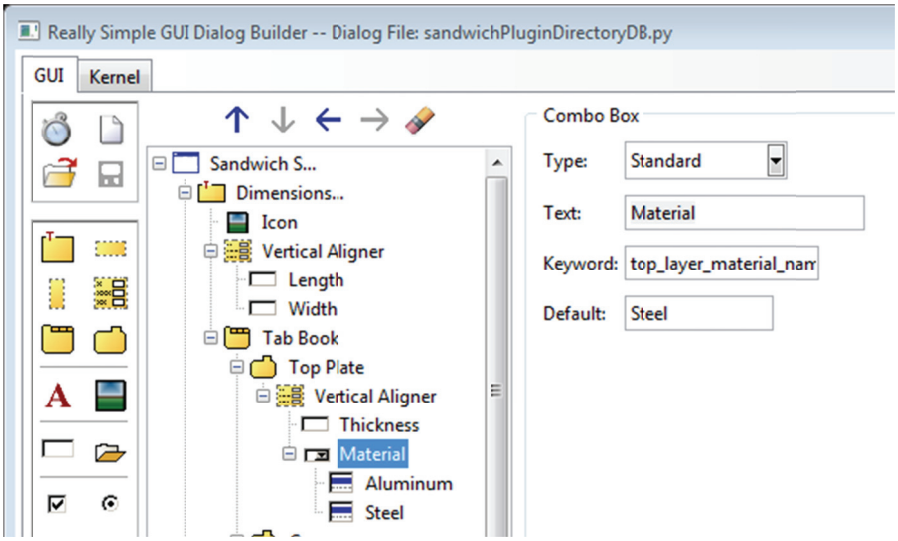


A vertical aligner is used to position the widgets inside the top plate tab.

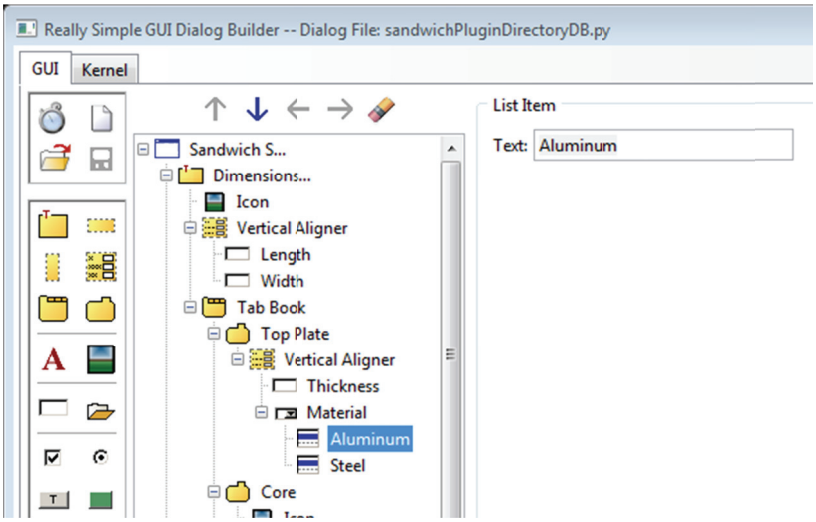


The text field 'Thickness' specifies the thickness of the top plate of the sandwich structure and is assigned the keyword **top_layer_thickness**.

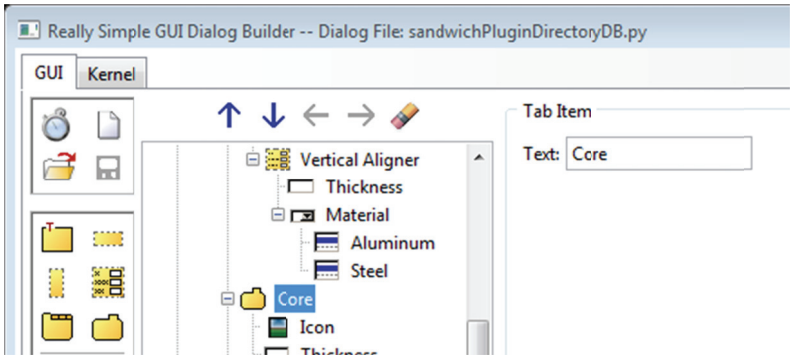




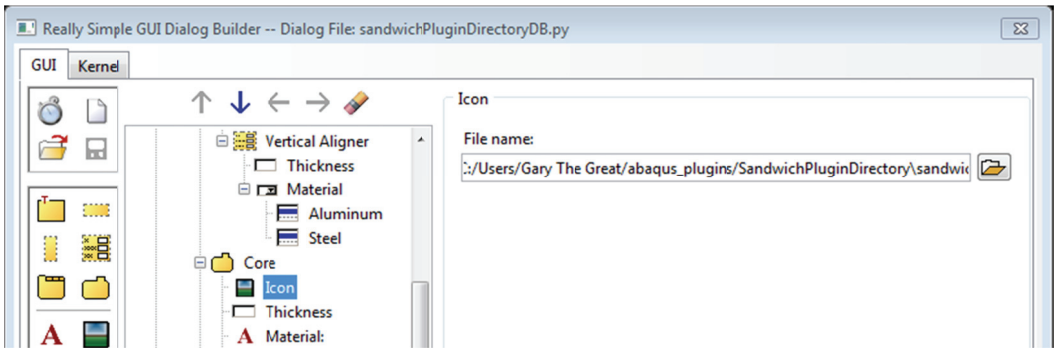
A standard combo box named 'Material' is created here. It is assigned the keyword **top_layer_material_name**. The default value has been set to 'Steel' which is one of the combo box items. Notice that the default value has been spelt exactly as the name of the combo box item 'Steel'. If you were to type anything other than 'Aluminum' or 'Steel' in the default field, it would be meaningless to Abaqus.



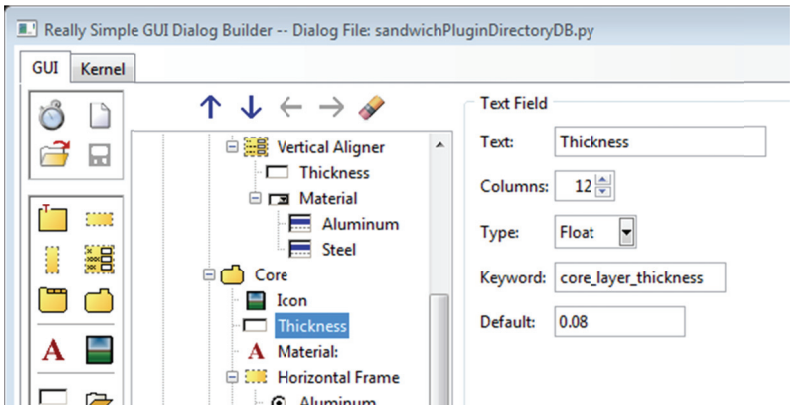
A combo box item 'Aluminum' is added here, followed by one named 'Steel'.



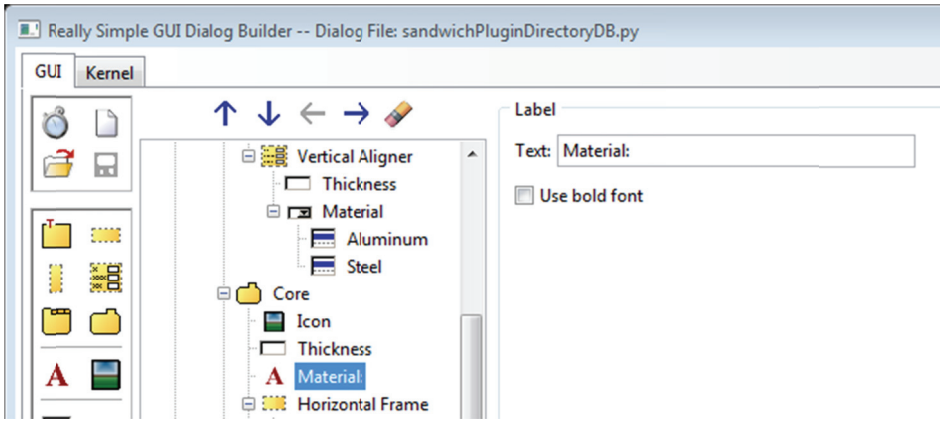
The second tab is named 'Core' and the user will define the properties of the core here.



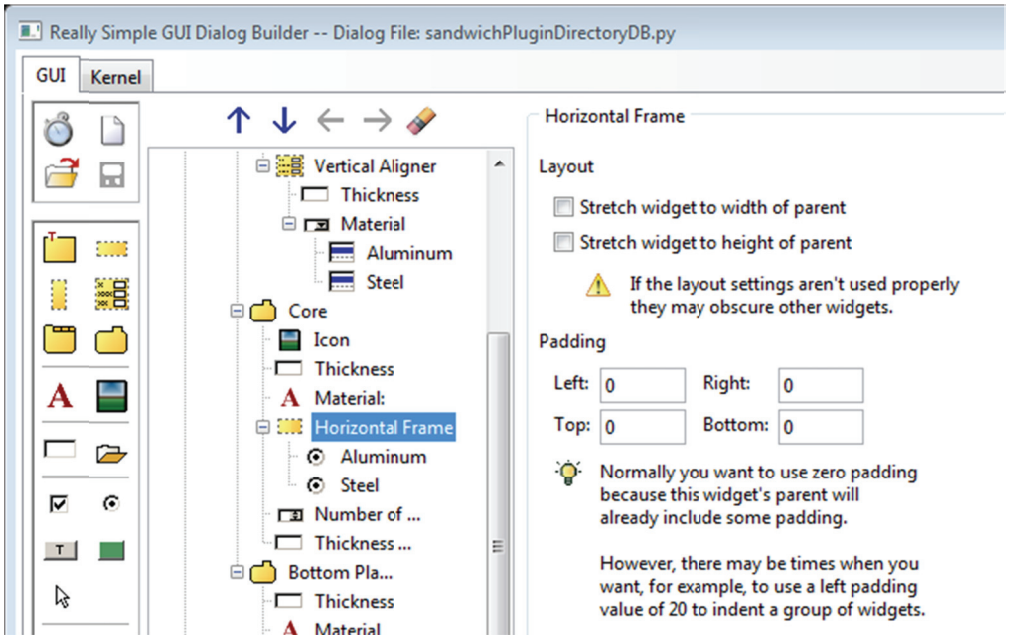
The icon widget is used to place an image of the core in the core tab.



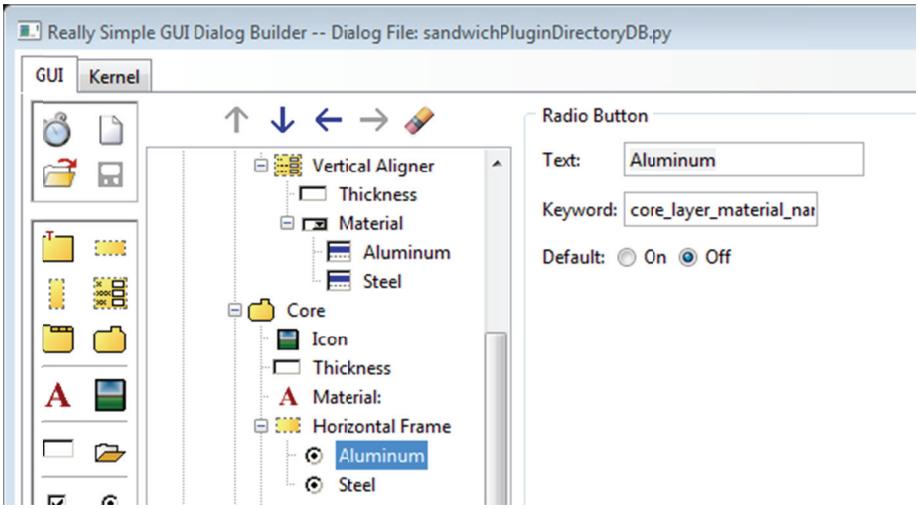
A text field labeled thickness is created and assigned the keyword **core_layer_thickness** and a default value of 0.08.



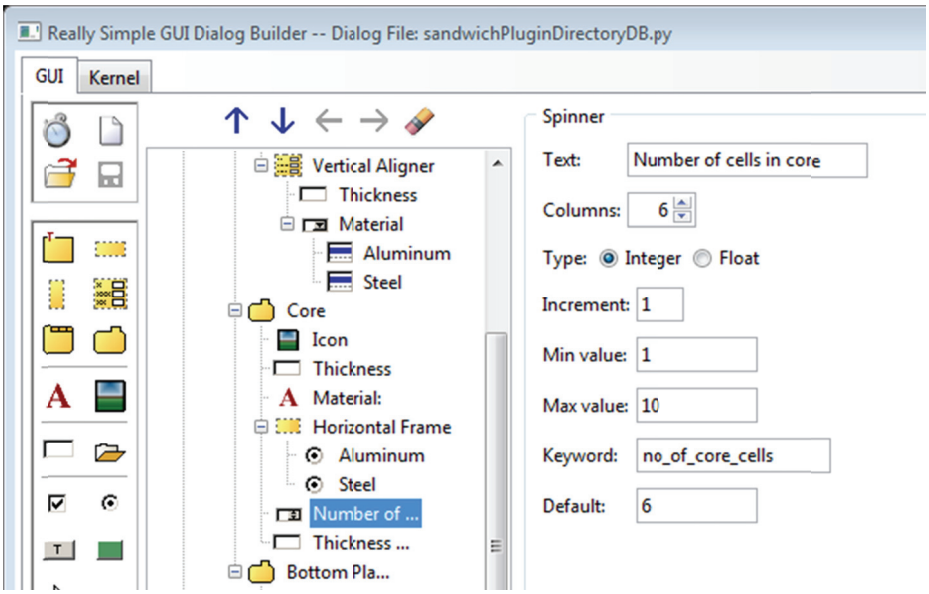
A read only text label with the text 'Material' is added to the core tab.



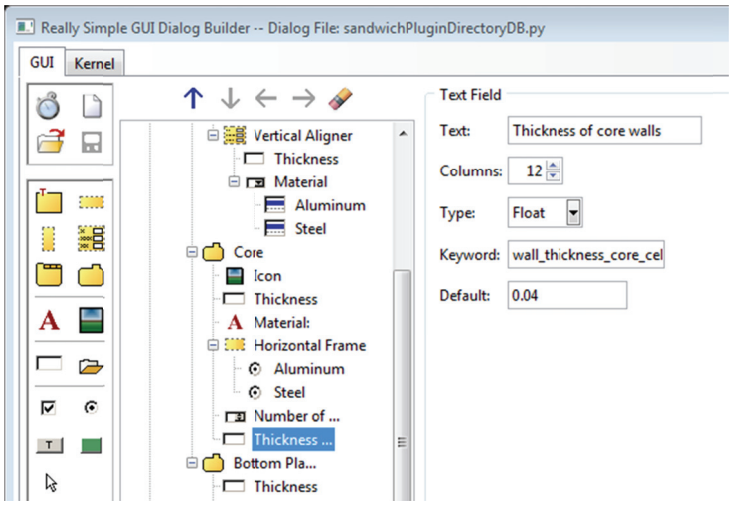
A horizontal frame is created in which we will place the radio buttons for the two materials. This will make them appear side by side.



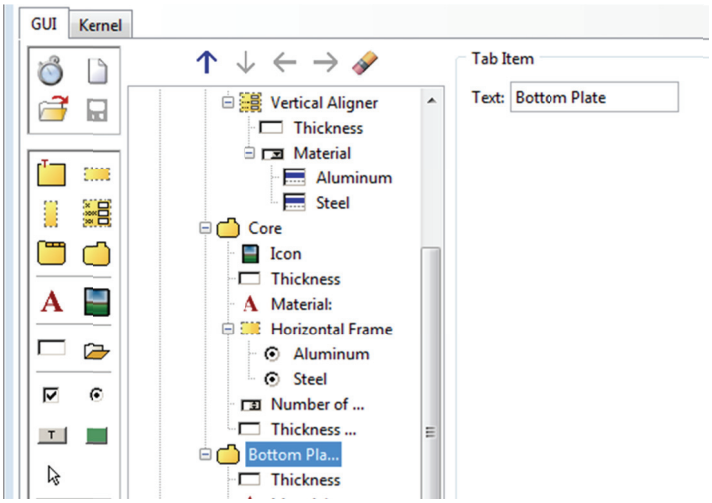
Radio buttons are created for 'Aluminum' and 'Steel'. Radio buttons allow you to select just one out of a set of options. If you select one radio button, the other will get deselected. In order to enforce this behavior, both radio buttons must be given the same keyword **core_layer_material_name**. If they are given different keywords they will not be part of the same radio group and will operate independently, meaning that you will be able to select both of them at the same time which will be quite meaningless.



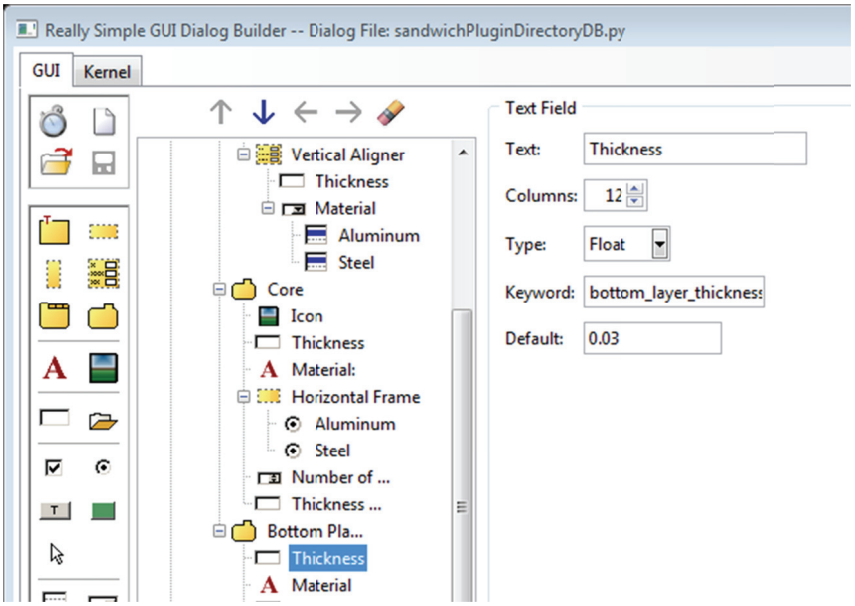
A spinner is used to allow the user to select the number of cells in the core. It is given the label text 'Number of cells in core' which will appear next to it in the dialog box. It will allow the user to select a value between the specified minimum of 1 and the specified maximum which is 10. The default has been set to 6. The selected value will be passed to the parameter **no_of_core_cells**.



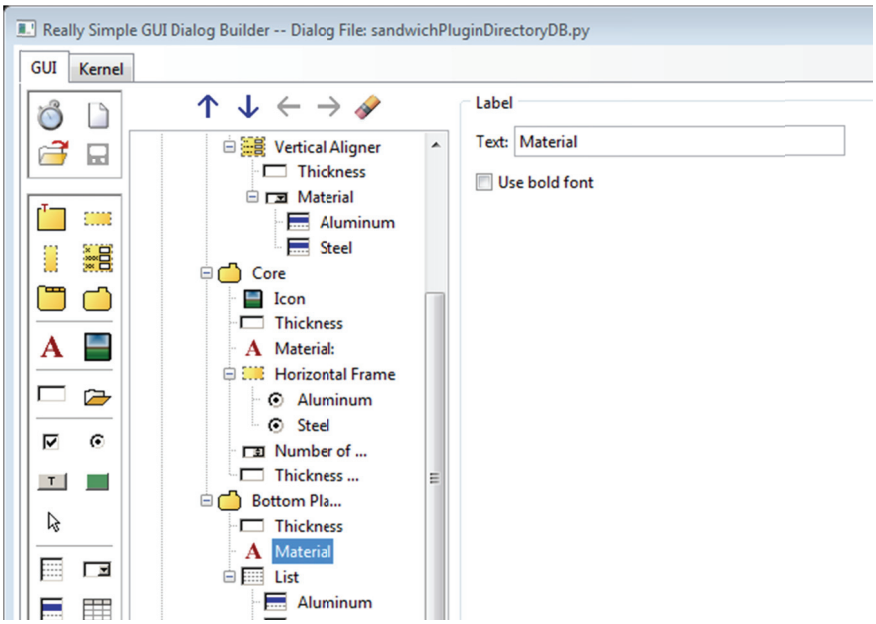
A text field is supplied for the user to enter the thickness of the walls of the core cells.



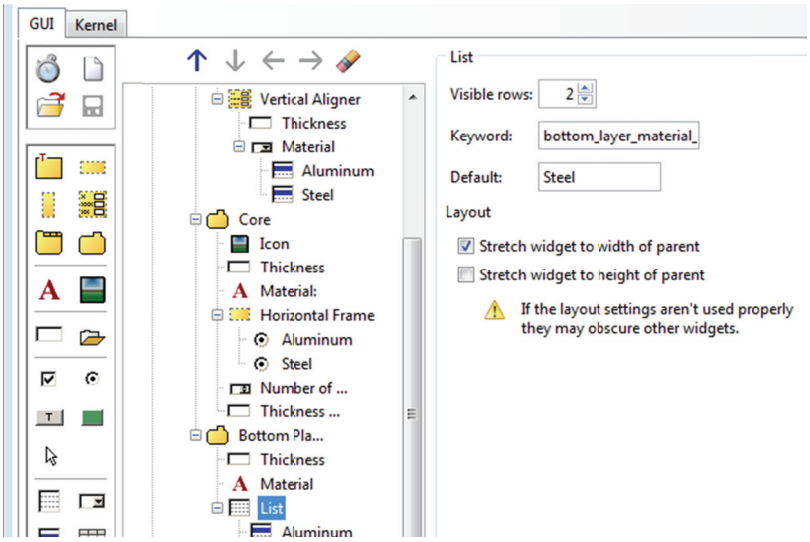
The third tab is named 'Bottom Plate'.



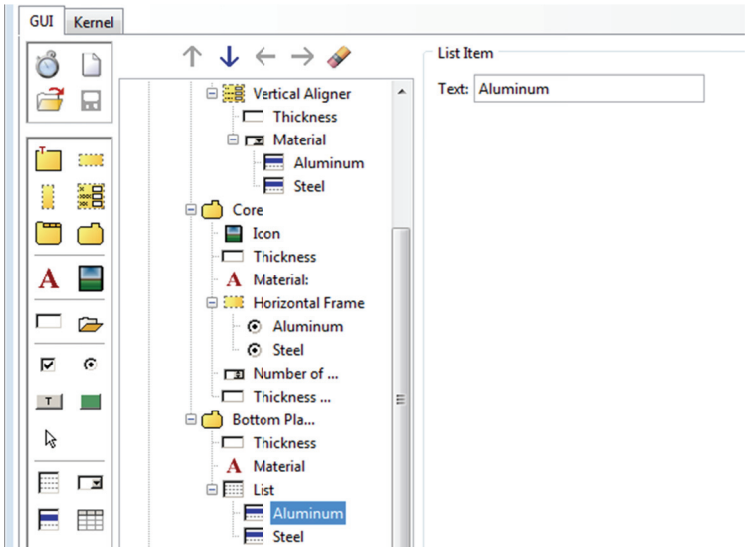
A text field is supplied for the user to enter the thickness of the bottom layer.



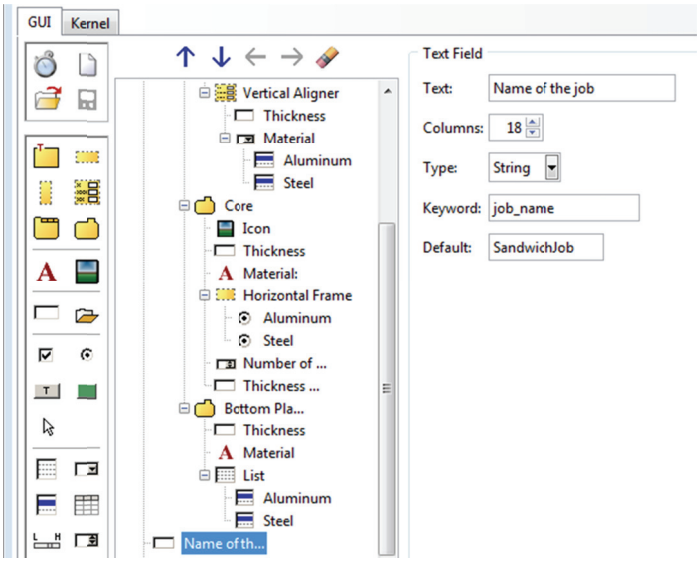
A text label ‘Material’ is inserted on the canvas.



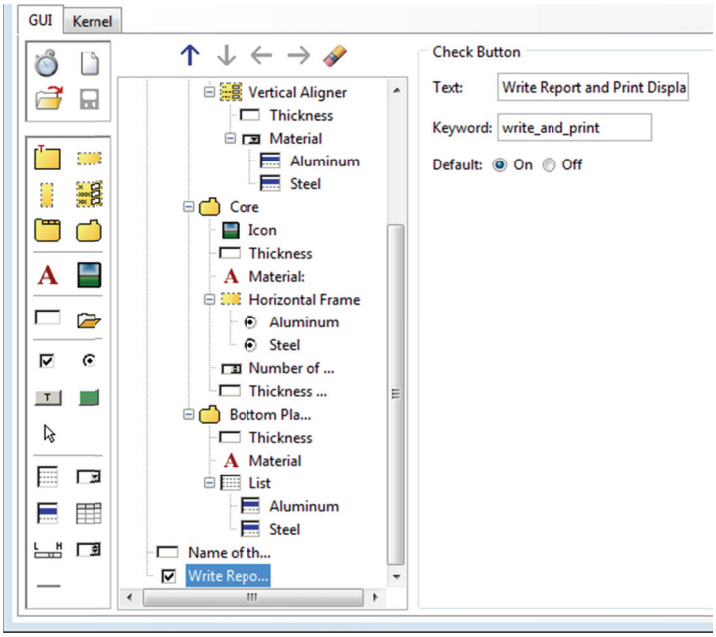
A list is used to provide the user with material options. The keyword **bottom_layer_material_name** is applied to the list container itself rather than individual list items. The default is set to 'Steel' which is one of the list items. Note that the default must be a name of one of the list items, in this case 'Aluminum' or 'Steel' otherwise it would be meaningless.



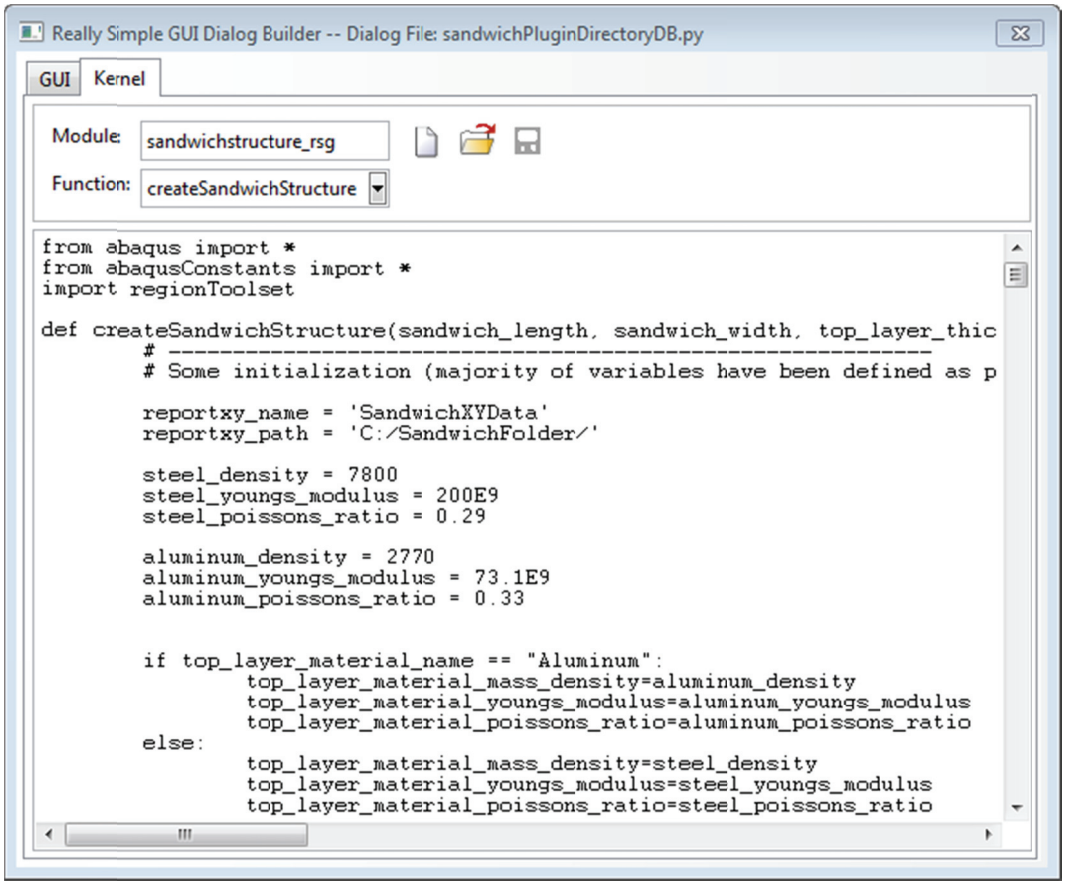
List items 'Aluminum' and 'Steel' are added to the list container.



A text field is provided for the user to supply the job name.



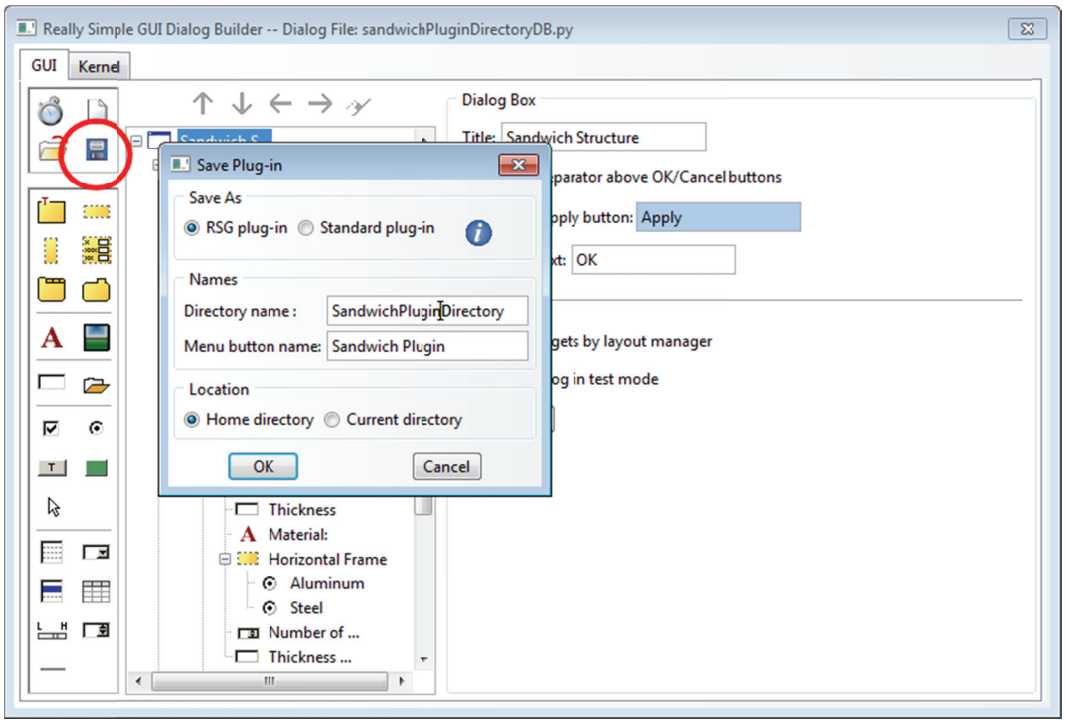
A checkbox allows the user to specify whether or not the XY report should be written and the displacement subsequently printed to the message area.



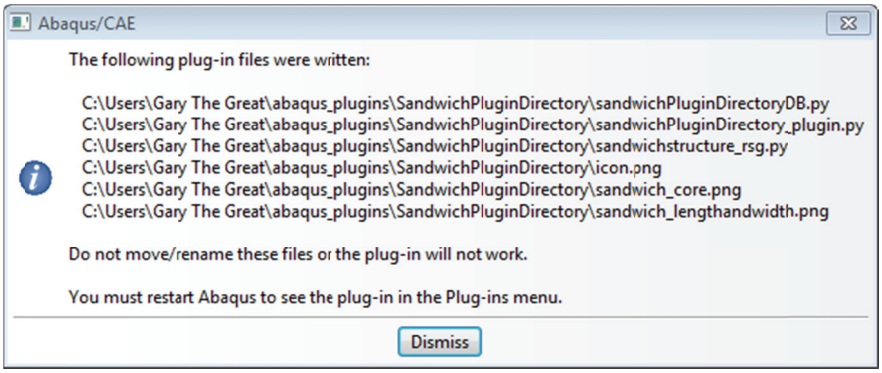
In the **Kernel** tab, we set the module to 'sandwichstructure_rsg' and the function to 'createSandwichStructure'. This means our script will be in the file **sandwichstructure_rsg.py** and will contain a function called **createSandwichStructure()**.

We now save the RSG Dialog Box as a plug-in by clicking the 'Save your dialog box as a plug-in' button. We shall save it as an RSG plug-in, which means internally Abaqus will use RSG commands to construct it. If we were to save it as a standard plug-in, Abaqus would use the GUI toolkit commands instead. You will learn about those in the next two chapters. We set the location to 'Home directory' which tells Abaqus to save the plug-in in the default plug-ins folder. On my Windows 7 system this is C:\users\(username)\abaqus_plugins\. The directory name is the name of the directory in

which the scripts will be stored – these scripts include the RSG plug-in startup, and RSG dialog construction scripts generated by Abaqus, as well as the kernel script written by us. The menu button name specified by you will be the name of the plug-in in the Plug-ins menu in Abaqus/CAE. Note that it will only be visible in the Plug-ins menu after you restart Abaqus/CAE.



When you click OK Abaqus will inform you of which files were saved and where. Since we selected 'Home directory' these are saved in the 'abaqus_plugins' folder.



19.5 Python Script to respond to the GUI dialog inputs

(Section removed from preview)

19.6 Examining the Script

(Section removed from preview)

19.7 Summary

In this chapter, you discovered that the RSG is, as its name suggests, “really simple”. You can rapidly create a dialog box with useful widgets, and hook it up to a kernel script. This script needs to have a function that accepts the data from the widgets as inputs. The RSG is suitable for a simple GUI interfaces, and the fact that it gets stored as a Plug-in makes it accessible within all instances of Abaqus/CAE.

20

Create a Custom GUI Application Template

20.1 Introduction

GUI Customization allows Abaqus users to modify or customize the Abaqus/CAE Interface. The analyst can change the look and feel of Abaqus/CAE to a great extent, creating his own modules, menus, toolbars, tool buttons and dialog boxes. He can also remove existing Abaqus/CAE modules and toolsets.

This technology has many uses. Think of a company or research institute that, for the most part, runs a handful of analyses on a regular basis with minor changes to these. A vertical application can be built with much of the repetitive tasks automated with scripts, giving the analyst the ability to make only certain allowed changes, and automating the rest of the process. This type of automation of in-house processes is of great use to some organizations.

This may be compounded by the fact that a lot of the personnel working on a project are not very proficient at using Abaqus, but need to harness its functionality and run simulations within a narrow framework. An application can be created which guides them through the process step by step, prompting them for inputs and hiding most of complexity of the Abaqus interface from them.

GUI Customization does not require an entire automated application to be built, it can be used to create plug-ins which accomplish a single specific task and have a well designed interactive interface suited to this.

You need to understand the fundamentals of Abaqus GUI development before we attempt to write a script. It is important that you read the following sections and understand them before we get into our GUI example.

20.2 What is the Abaqus GUI Toolkit

Abaqus extends the functionality of a 3rd party open source GUI toolkit called the FOX toolkit. FOX is a cross platform C++ based toolkit for creating GUIs. If you wish to learn more about this toolkit you can visit their website at http://www.fox-toolkit.org/.

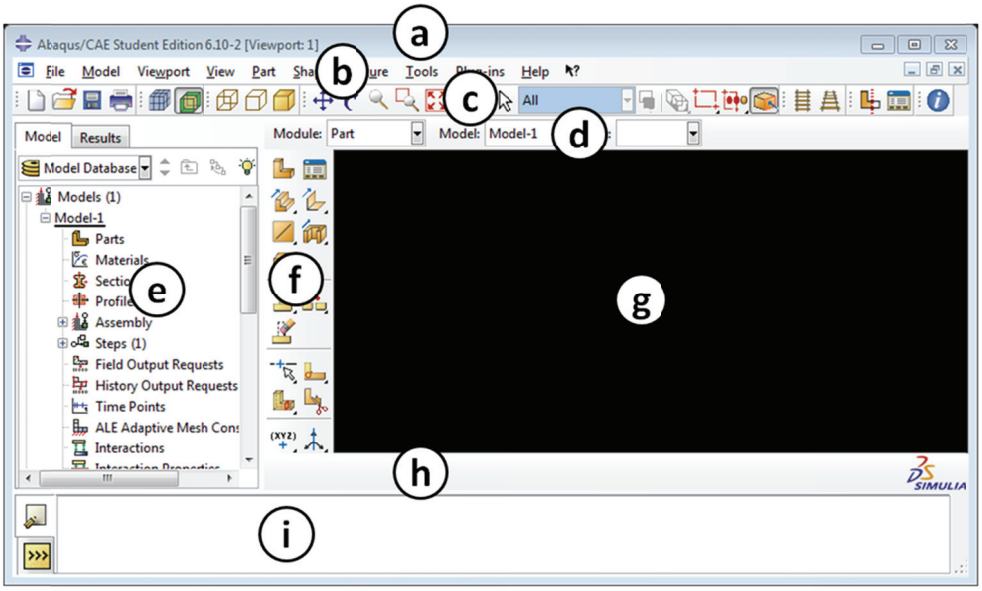
Abaqus provides a Python interface to the Abaqus/CAE C++ GUI toolkit. This interface, or toolkit, is called the Abaqus GUI Toolkit.

20.3 Components of a GUI Application

In order to design an Abaqus GUI Application it is very important that you understand the GUI infrastructure - the components that constitute the GUI, and how they work together.

1. The top most component is the application object itself. This is an object of type **AFXApp** which you will learn more about in a little bit.
2. The application consists of a window with the GUI infrastructure. All custom Abaqus applications have this basic look. The window consists of
   a) a title bar,
   b) a menu bar,
   c) one or more toolbars,
   d) a context bar which consists of the module control and context controls
   e) a tree area which displays the model tree or output database tree
   f) a module toolbox with tool buttons
   g) a canvas area where the parts, assemblies, renderings and so on are displayed
   h) a prompt area below the window
   i) and a message area (which can be switched with the command line interface)

   These are marked in the figure. The main window itself is an object of type **AFXMainWindow**.



3. Within the main window you have modules and toolsets. Modules are clearly marked in Abaqus/CAE with the word "Module:" and a combo box (drop down menu) listing the different modules such as Part, Property, Assembly, Step, Visualization and so on. This combo box is visible in the context bar (d) in the figure. Modules are of type **AFXModuleGui**. Toolsets on the other hand are the buttons displayed right next to the canvas in the same area as module toolboxes (f). However they are different from module toolboxes in that module toolboxes change depending on which module you are in whereas toolsets remain there no matter which module you are in. Toolsets are of type **AFXToolsetGui**.
4. Within the modules you have menus, toolbars and module toolboxes. As you switch modules, these change. Menus have panes which are of type **AFXMenuPane**, and within these you have the menu title **AFXMenuTitle** and menu items **AFXMenuCommand**. Toolbars exist as groups of type **AFXToolbarGroup** and they are made up of toolbar buttons of type **AFXToolButton**. Toolboxes also exist as groups of type **AFXToolboxGroup** and these consist of toolbox buttons **AFXToolButton** similar to toolbars.
5. The menus, toolbar buttons and toolboxes launch modes. Modes get input from the user and issue a command. There are two types of modes – form modes and procedure modes.

Form modes create a dialog box where the user can type in inputs or select options using checkboxes, radio buttons, lists and so on. For example, when you click on **View > Part Display Options**, you see the **Part Display Options** dialog box. You can select your options here and when you click **Apply** a command is issued to the kernel. Form modes do not allow the user to pick anything in the viewport. Form modes are of type **AFXForm**.

Procedure modes on the other hand prompt users to make selections in the viewport and then use this information to execute a kernel command. So for example, if you try to define a concentrated force in the loads module, Abaqus prompts you to select the nodes on which to apply it and you pick the nodes in the viewport window. This is a procedure mode. Procedure modes can have multiple steps. They can also be used to launch dialog boxes. Procedure modes are of type **AFXProcedure**. It is also possible for menu items, toolbar buttons or toolbox buttons to launch a dialog box that is not associated with a form or procedure. This type of dialog will not communicate with the kernel, only with the GUI (more on this later). Such a dialog box will be of type **AFXDialog**.

6. Form modes launch dialog boxes of type **AFXDataDialog**. These are different from the previously mentioned **AFXDialog** because **AFXDataDialog** dialog boxes send commands to the kernel for processing. Procedure modes create objects of type **AFXPickStep** and can also launch dialog boxes of type **AFXDataDialog**.
7. Dialog boxes are made up of layout managers such as **AFXVerticalAligner** which creates a vertical layout, and many others which we shall discuss later.
8. The layout managers contain within them the widgets such as labels (**FXLabel**), text fields (**AFXTextField**), radio buttons (**FXRadioButton**) and so on.

It is important that you understand the above structure and recognize the names of the classes. Scripts written to target the Abaqus GUI Toolkit usually span multiple .py files and it can get a little confusing to keep track of what goes where if you don't fully understand the structure.

20.4 GUI and Kernel Processes

In the previous section we mentioned **AFXDialog** and **AFXDataDialog**, and briefly spoke of how one (the second one) sends commands to the kernel while the other (the first one) does not. It is important to understand that when you create a custom Abaqus GUI, you have two types of processes running simultaneously – GUI processes and

kernel processes. GUI processes execute GUI commands and kernel processes execute kernel commands.

You've already seen kernel commands. All of the scripts written up until this point were kernel scripts. They interacted with the Abaqus kernel in order to set up your model, send it to the solver, and post process it. To elaborate further, only a kernel script can have a statement such as

```
mdb.Model(name=My Model, modelType=STANDARD_EXPLICIT)
```

or

```
myPart = myModel.Part(name='Plate', dimensionality=THREE_D, type = DEFORMABLE_BODY)
```

Model() and **Part()** are commands that are executed by the Abaqus kernel. Kernel scripts usually have the following import statements at the top

```
from abaqus import *
from abaqusConstants import *
```

GUI scripts on the other hand only deal with GUI processing. They create the GUI, and can issue Python commands, but not commands that target the Abaqus kernel. They usually have the import statement

```
from abaqusGui import *
```

at the top.

GUI and kernel scripts must be kept separate. You cannot have "from abaqus import *" and "from abaqusGui import *" in the same script as a script must either be purely GUI or purely kernel.

Since the GUI must eventually issue commands to the kernel, a link must be established between GUI and kernel scripts. This is usually done using a mode. For example, a form mode (**AFXForm**) launches a dialog (**AFXDialog**) which contains the GUI commands necessary to display widgets (checkboxes, text fields, labels etc), and when the **OK** button is pressed in the dialog box the form calls a command in a separate kernel script. This way the GUI and kernel scripts are kept separate and one calls the other through the use of a mode. Another method is to use **sendCommand()** method. You will see both of this demonstrated in the next chapter, but it is essential that you learn these concepts right now.

20.5 Methodology

In this example we create a basic GUI application. As such it does not execute any kernel scripts; it is just a GUI with no real functionality. However it is a complete framework, and we will be using it for the example in the next chapter. More importantly, this code framework can be reused by you in all GUI scripts you write in the future, as it serves as a stable base off which you can build.

The GUI application is created using a number of scripts. We will examine each of these scripts in turn, but first an overview so that you see the bigger picture.

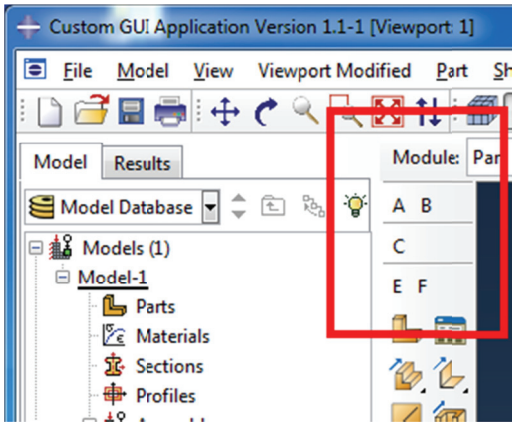
- **customCaeApp.py** is the application startup script. It creates the application (**AFXApp**) and calls the main window
- **customCaeMainWindow.py** creates the main window (**AFXMainWidnow**). It registers the toolsets and modules that will be part of the application. These toolsets and modules include standard ones as well as custom ones made by us.
- **modifiedCanvasToolsetGui.py** creates a modified version of the **Viewport** menu which you see when you open Abaqus/CAE. It will adds a few new menu items to the **Viewport** menu, removes others that exist by default, adds a couple of horizontal separators in the menu pane, and changes the name of the **Viewport** menu to 'Viewport Modified'.
  When menu items or toolbar buttons are clicked in this modified viewport toolset, the form mode, defined in **demoForm.py**, is called to post the dialog box which is defined in **demoDB.py**
- **customToolboxButtonsGui.py** creates a new toolset (**AFXToolsetGui**). The toolset buttons which appear to the left of the canvas (along with module toolboxes) will be visible in all modules.
  When buttons in this toolbox are clicked, the form mode defined in **demoForm.py** is called to post the dialog box defined in **demoDB.py**
- **customModuleGui.py** creates a new module (**AFXModuleGui**) which appears in the module combobox as 'Custom Module'. This module has a menu (**AFXMenuPane**) called 'Custom Menu' associated with it, a toolbar (**AFXToolbarGroup**) called 'Arrow Toolbar' and a toolbox group (**AFXToolboxGroup**). All of these are only visible when the user is in the custom module.

When most of the menu items, toolbar buttons or toolbox buttons are clicked in this custom module, the form mode defined in **demoForm.py** is called to post the dialog box defined in **demoDB.py**. However to change things up, one of the menu items instead posts a modeless dialog defined in **demoDBwoForm.py** without calling any form mode. This is to demonstrate how you launch a modeless dialog box.

- **demoForm.py** creates a form mode (**AFXForm**) which will post the dialog created in **demoDB.py** and will issue a command when the **OK** button is clicked in that dialog.
- **demoDB.py** creates the modal dialog box (**AFXDataDialog**) that will be posted by the form mode of **demoForm.py**
- **demoDBwoForm.py** creates a modeless dialog box – one that is posted without any form.

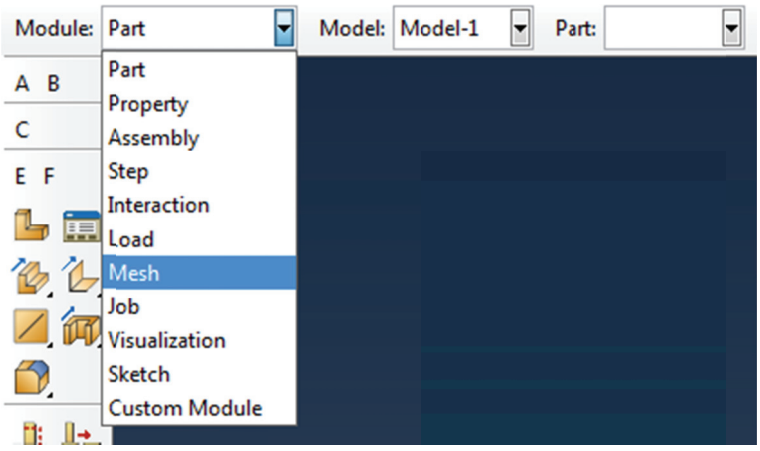
20.6 Python Script

We shall now look at each of the script files in turn. Remember that these must all exist together in the same folder for the application to work.

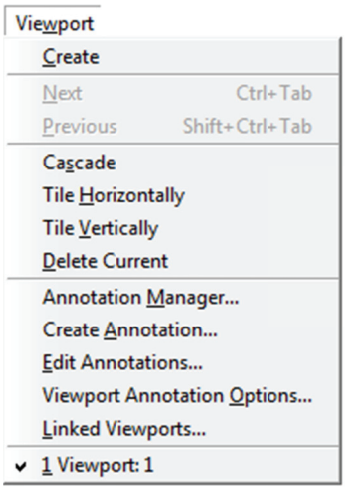


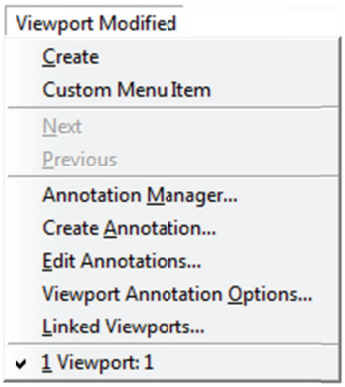
(Contents removed from preview)

(Contents removed from preview)

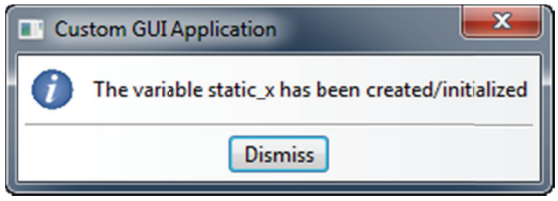


(Contents removed from preview)

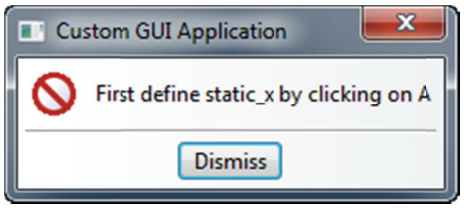




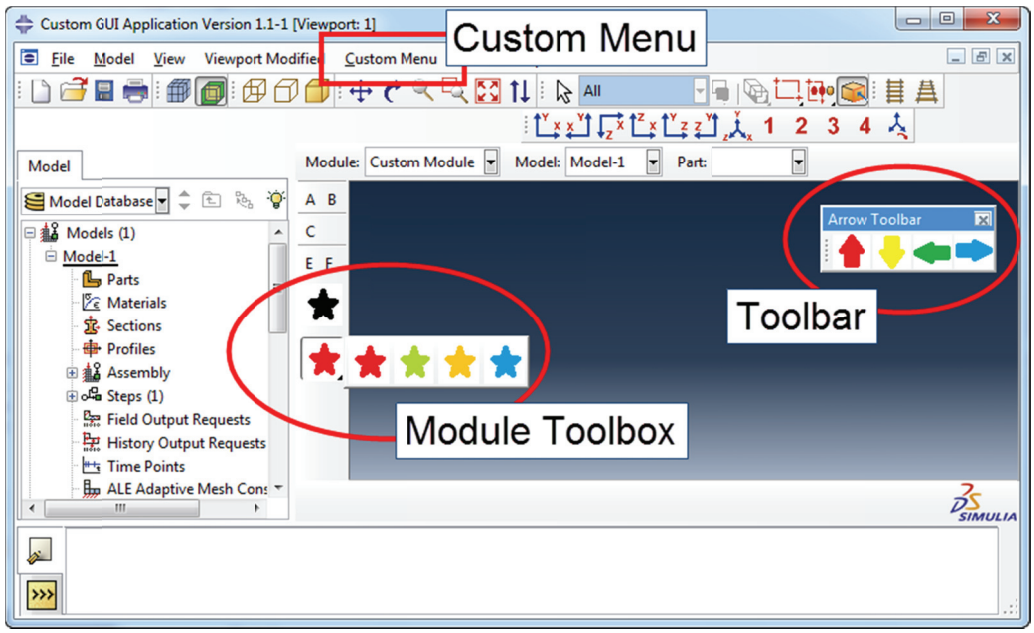
(Contents removed from preview)

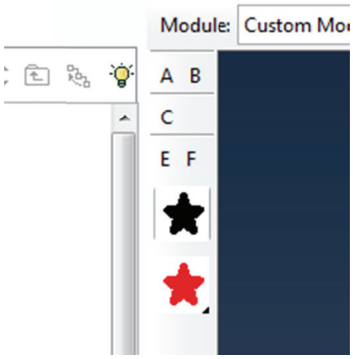


(Contents removed from preview)



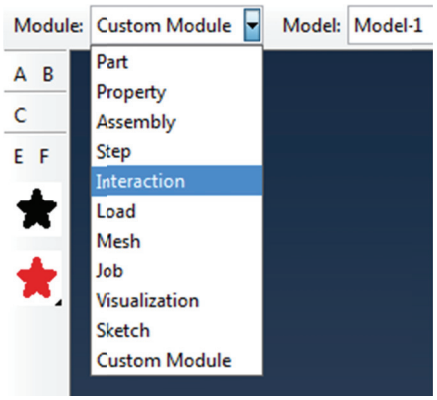
(Contents removed from preview)



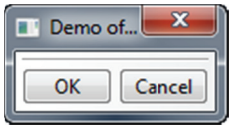


(Contents removed from preview)

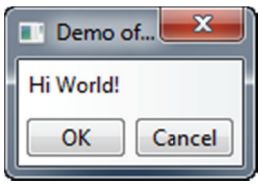
(Contents removed from preview)



(Contents removed from preview)



(Contents removed from preview)



(Contents removed from preview)

20.7 Summary

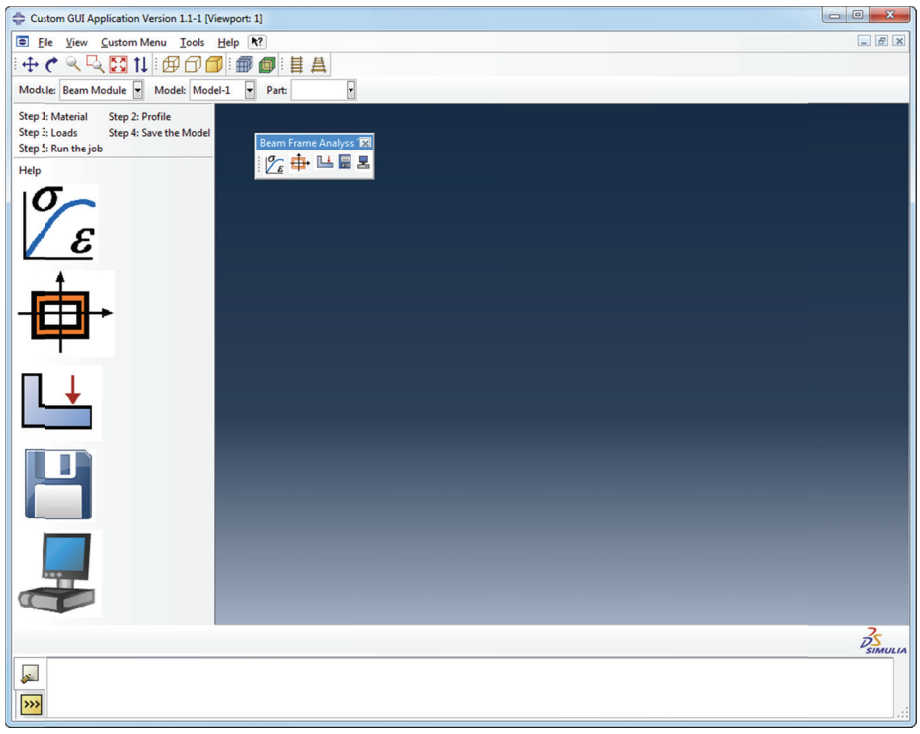
We created a working GUI framework in this chapter in order to explain the process of writing the scripts, and also to understand the inner workings of the Abaqus GUI infrastructure. The application created here does not do anything useful on its own, however the basic framework has been created, and it is one you can reuse when creating your own GUI applications. In fact we shall reuse it in the next chapter.

21

Custom GUI Application for Beam Frame Analysis

21.1 Introduction

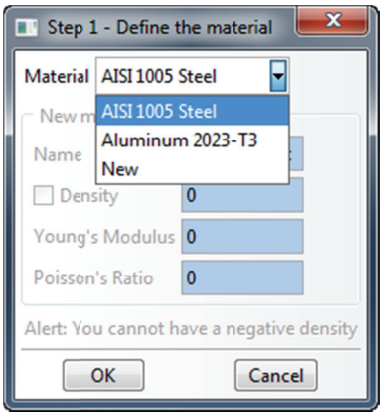
In the previous chapter we created a framework that can be reused for any GUI application. It included a persistent toolset, a custom module with menus, toolboxes, toolbuttons and a toolbar, and other customizations to the standard GUI interface.



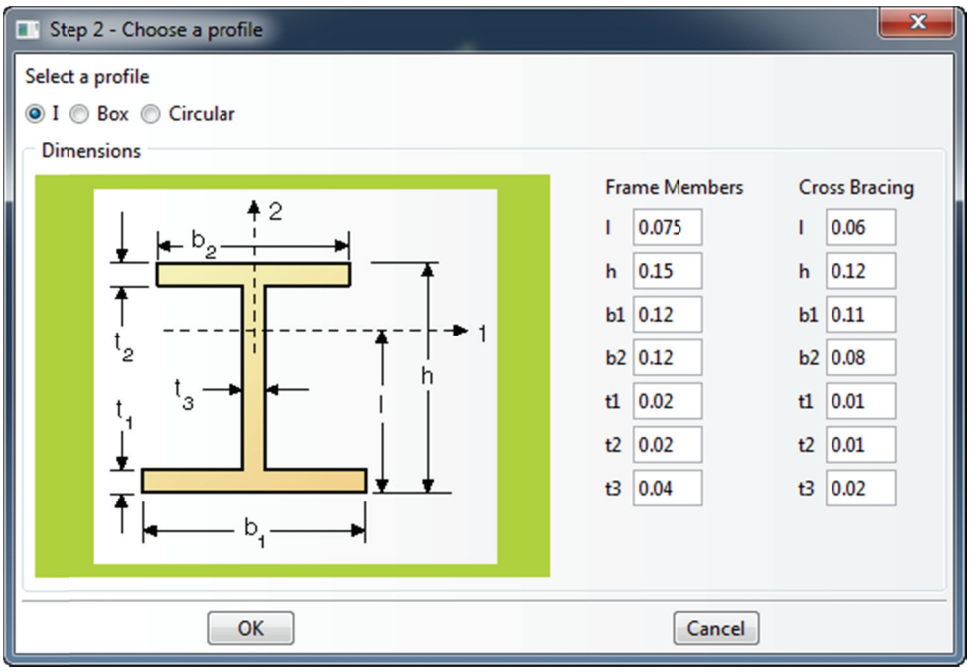
In this chapter we will create a functional application that demonstrates project automation. We will use the beam frame model from Chapter 9. The application will create this same beam frame simulation, but prompt the user for inputs along the way. It will create a custom interface where the user can only perform certain actions, and only when prompted to do so, just as you would expect from a vertical application.

The figure displays our custom GUI application. It will not have a model tree on the left. The majority of menus and toolbars are removed leaving only a few barebones items. There is a persistent toolset with buttons 'Step 1' thru 'Step 5'. All the modules are removed as well leaving only a custom module called 'Beam Module'. This module has a module toolset which consists of 5 large buttons (with large icons on them). A custom toolbar is available with buttons and small icons. There is also a menu called 'Custom menu' with 5 menu items. The persistent toolset, beam module toolset (with the big icons), the toolbar, and the custom menu all have 5 buttons/items and provide the exact same functionality.

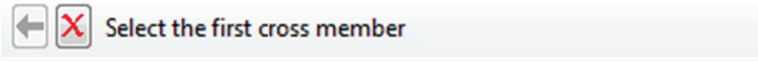
When 'Step 1' is initiated using any of the buttons or menus, the user is prompted for material properties. He can select 'Steel' or 'Aluminum' or define a new material. When the user clicks **OK**, Abaqus proceeds to create the model, beam parts (frame and crossbracing) and materials (using the users input).



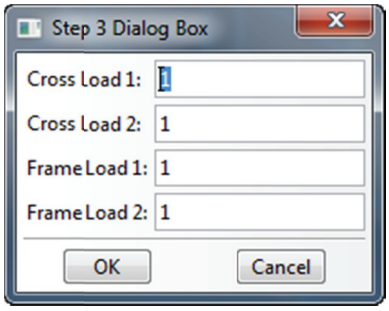
When 'Step 2' is initiated, the user is prompted to create the profile of the beam with options of 'I', 'Box' and 'Circular'. A number of default values are filled into the fields which the user can alter. When the use clicks **OK** the profiles are created. The application also proceeds to create the sections and assembly.



When 'Step 3' is initiated, the user is prompted to select a cross member, then a second, and then two frame members. The user will be able to pick these in the viewport.

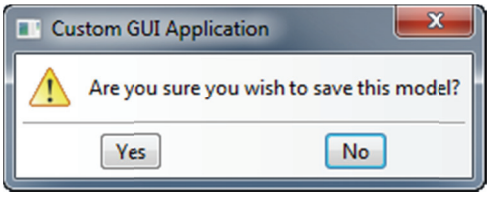


The application will then prompt the user to enter loads for each of the members selected.

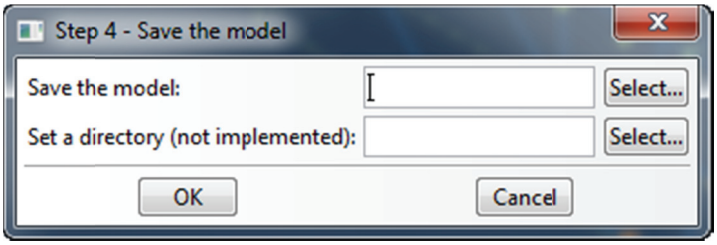


On accepting these inputs, the application will create the loads and display the assembly with loads in the viewport.

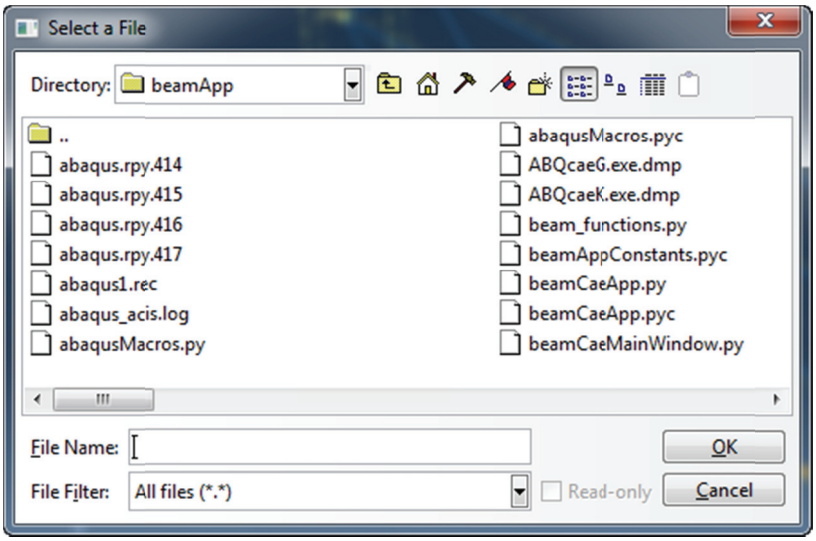
'Step 4' asks the user if he wishes to save the model'.



If he clicks **Yes** he is asked to provide a path at which to save the model.

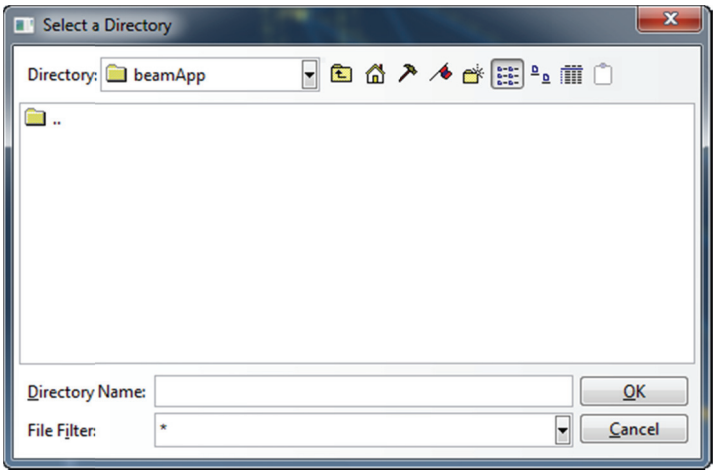


If he clicks the **Select…** button, he will be provided a file selection window



The directory selection on the other hand is not actually implemented in this application, but is provided to show you how to present the user with a directory selection window if

you need to do so in one of your own scripts. If the user clicks **Select…** next to 'set a directory', he will see the directory selection window.



When the **OK** button is finally clicked, the entire model is saved at the specified file location.

Finally 'Step 5' runs the analysis.

21.2 Layout Managers and Widgets

In the custom CAE example of the previous chapter, our dialog boxes were mostly empty. This time they will be populated with useful text fields, check boxes, radio buttons and combo boxes. All of these are known as widgets. In fact regular buttons, toolbar and toolbox buttons, flyout buttons and menu buttons are also widgets, so you have in fact used widgets before. Widget is a generic term for GUI controls, and these widgets allow a user to interact with the program.

Layout managers are containers used to arrange widgets in a dialog box. You place the widgets within the layout manager, and depending on the type of layout manager those widgets will be placed in an ordered manner in the dialog box. For example, a vertical alignment layout manager will cause all widgets inside it to be placed one below the other. A tab book layout manager on the other hand will allow you to have multiple tabs, and different widgets in each tab which will be displayed only when the user is in that tab.

You'll use layout managers and widgets in the dialog boxes for 'Step 1' through 'Step 4' so you'll have a good understanding of them by the end of the chapter.

21.3 Transitions and Process Updates

Transitions allow you to detect changes in the state of widgets. The program can then change the GUI state in a dialog box based on the detected activity. For example, in the dialog box for 'Step 1', the user is presented with 3 material choices – 'AISI 1005 Steel', 'Aluminum 2024-T3' and 'New'. A transition is added to the application to detect whether the user has clicked 'New' or not, and if he has, a number of text fields are enabled allowing him to provide a name and material properties for this material. On the other hand if 'Steel' or 'Aluminum' are selected, these material property fields will be disabled or grayed out.

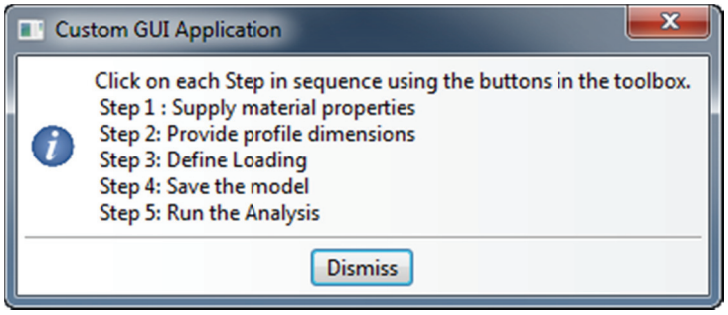
The transition allows the program to detect the change in state of the combo box widget and execute the appropriate method to enable or disable the text fields. Transitions do this by comparing the value of the keyword associated with the widget with a specified value and doing a simple comparison such as EQ (equals), GT (greater than) or LT (less than). However sometimes you may need to perform a more complicated comparison, or meet some more complex condition that cannot be represented using simple comparisions such as EQ, GT and LT. In that case you will need to use process updates.

The **processUpdates()** method is called during every GUI update cycle. You can place your own code in this method to test for some condition, and if some condition is met then you can execute the relevant methods. Needless to say this should be used with caution since it is called at every GUI update, and if you have a lot of time consuming code here you can slow your program down considerably.

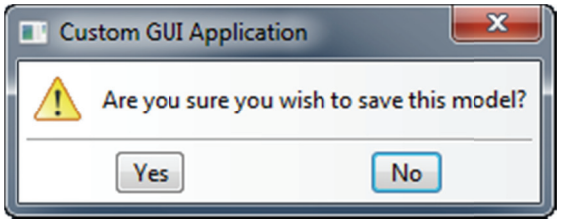
We will demonstrate how to use transitions in the dialog box for 'Step 1', and **processUpdates()** in the dialog box for 'Step 2'.

(Contents removed from preview)

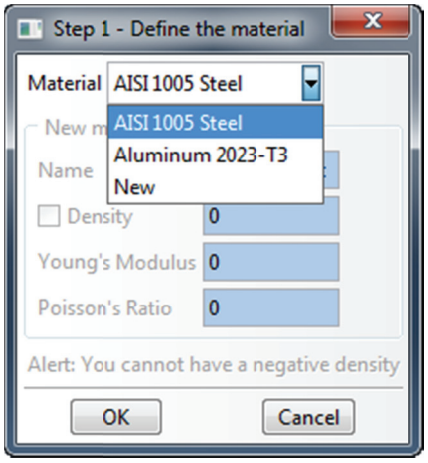
.



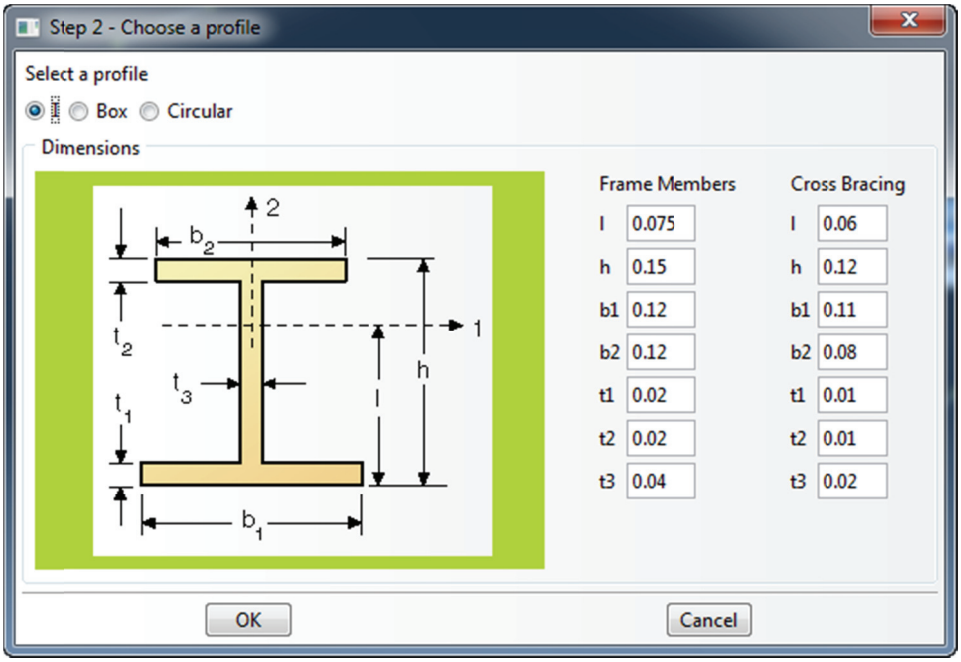
(Contents removed from preview)

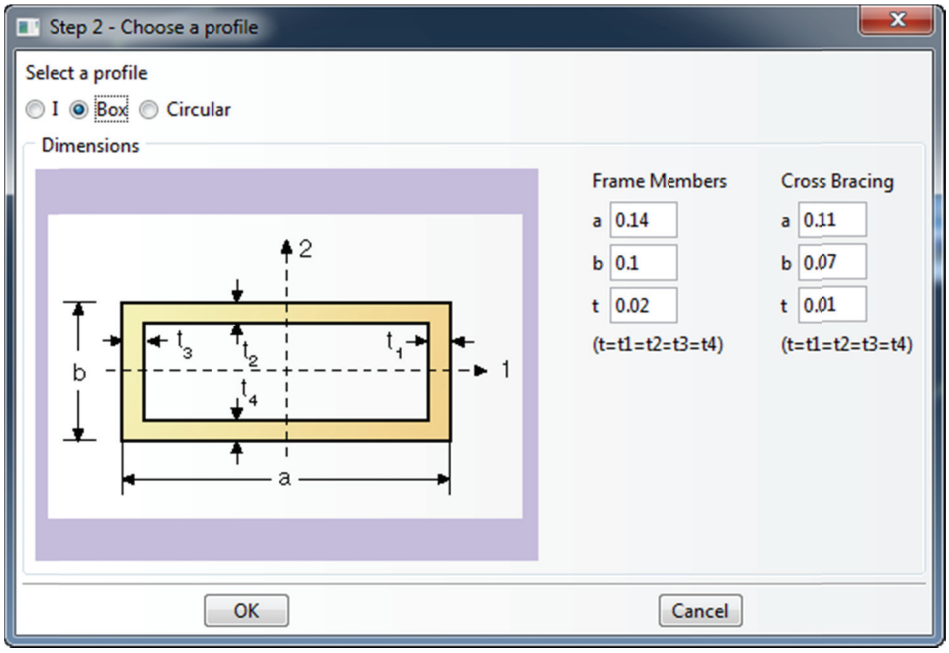


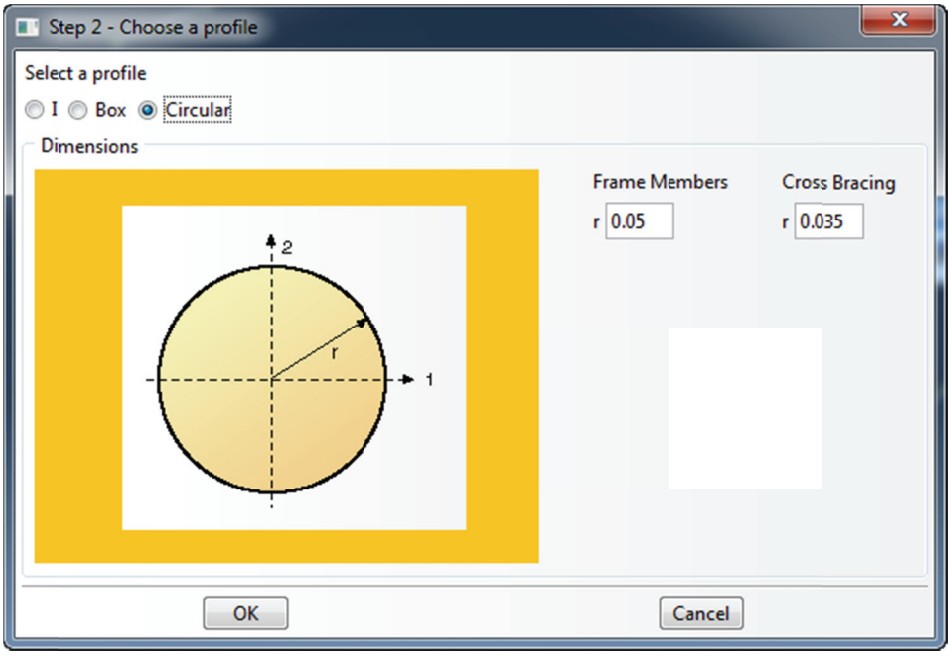
(Contents removed from preview)

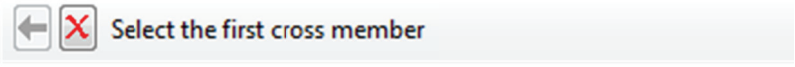


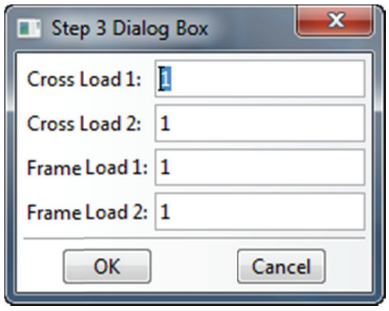
(Contents removed from preview)


Step 2 - Choose a profile
Select a profile
I
Box
Circular
Dimensions
2
b2
t2
t3
t1
1
h
l
b1
Frame Members
l 0.075
h 0.15
b1 0.12
b2 0.12
t1 0.02
t2 0.02
t3 0.04
Cross Bracing
l 0.06
h 0.12
b1 0.11
b2 0.08
t1 0.01
t2 0.01
t3 0.02
OK
Cancel


Step 2 - Choose a profile
Select a profile
I
Box
Circular
Dimensions
2
t3
t2
t1
b
1
t4
a
Frame Members
a 0.14
b 0.1
t 0.02
(t=t1=t2=t3=t4)
Cross Bracing
a 0.11
b 0.07
t 0.01
(t=t1=t2=t3=t4)
OK
Cancel







21.4 Summary

You've now created a fully functional custom GUI application and have a good understanding of the steps involved in scripting one. GUI design is a fairly complicated subject and you'll probably spend a lot of time debugging code, but hopefully the scripts from this chapter and the previous one will give you a great starting point for any GUI applications you develop.

Abaqus offers a number of widgets and layout managers aside from the ones used in this example so it is recommended that you take a look at the 'Abaqus GUI Toolkit User's Manual' and the 'Abaqus GUI Toolkit Reference Manual' for further information.

22

Plug-ins

22.1 Introduction

In this chapter we will talk about creating plug-ins. Plug-ins are scripts available to a user in Abaqus/CAE through the Plug-ins menu. They help extend the functionality of Abaqus. A plug-in can be a simple kernel script that performs a routine task, the same sort of script you could run through **File > Run Script…** In this scenario the advantage is that of convenience - the script is easily accessible to everyone who is using Abaqus/CAE once it is packaged as a plug-in. On the other hand the plug-in can be a GUI script which displays a custom interface prompting the user to input data and select items in the viewport. If all you need is a little extra functionality, creating a plug-in requires less work than writing an entire custom GUI application. However a plug-in cannot modify or remove Abaqus/CAE modules and toolsets the way a custom application can.

22.2 Methodology

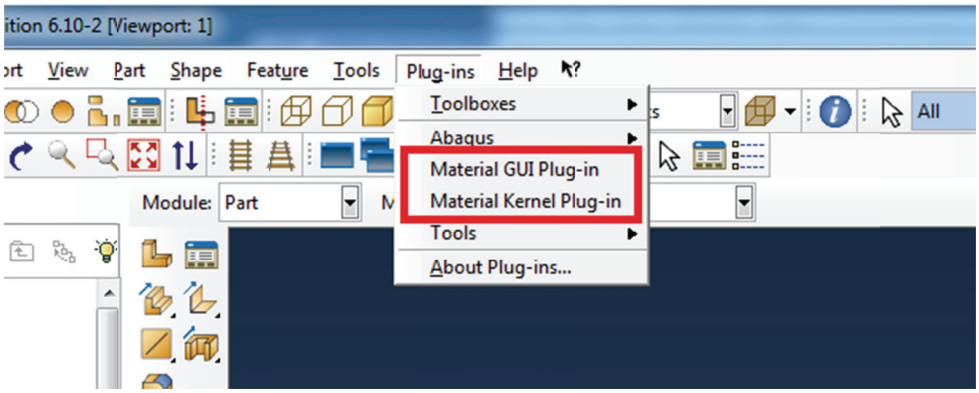
All plug-ins must follow the naming convention *_plugin.py. This helps Abaqus identify a script that is a plug-in. A plug-in may consist of more than one script; however the rest of the scripts do not need to follow this naming convention. Presumably your *_plugin.py script has **import** statements which will cause the other scripts to be imported as needed. Also, it is recommended that you store all these related scripts (and other files such as icons) in the same directory unless you wish to mess with the PYTHONPATH variable.

Abaqus/CAE automatically searches for plug-ins in certain directories while starting up. All plug-ins detected are added to the **Plug-ins** menu. Your plug-ins must be placed in one of these key locations. By default Abaqus searches for a folder called **abaqus_plugins**, first in the Abaqus directory (**abq_dir\cae\abaqus_plugins**), then the home directory (**home_dir\abaqus_plugins**), and finally the current directory (**cur_dir\abaqus_plugins**).

If a plug-in is a kernel plug-in, Abaqus/CAE sends commands of the form *module_name.function_name* to the kernel. If the plug-in is a GUI plug-in, Abaqus/CAE sends a command of the type **ID_ACTIVATE, SEL_COMMAND** to the GUI object created for the plug-in.

22.3 Learn by Example

Since kernel and GUI plug-ins operate slightly differently, we're going to create one of each. We shall call them 'Material Kernel Plug-in' and 'Material GUI Plug-in'. We won't write too much new code, we'll just reuse statements written in previous chapters and package them as plug-ins.



22.3.1 Kernel Plug-in Example

We will use the first script we wrote in this book, the one in Chapter 1, section 1.2. If you recall, all this script does is create 3 materials. We have placed it inside a function, **createMaterials()**, which our plug-in can call.

We place the contents in **materialkernelscript.py**. Here is the listing:

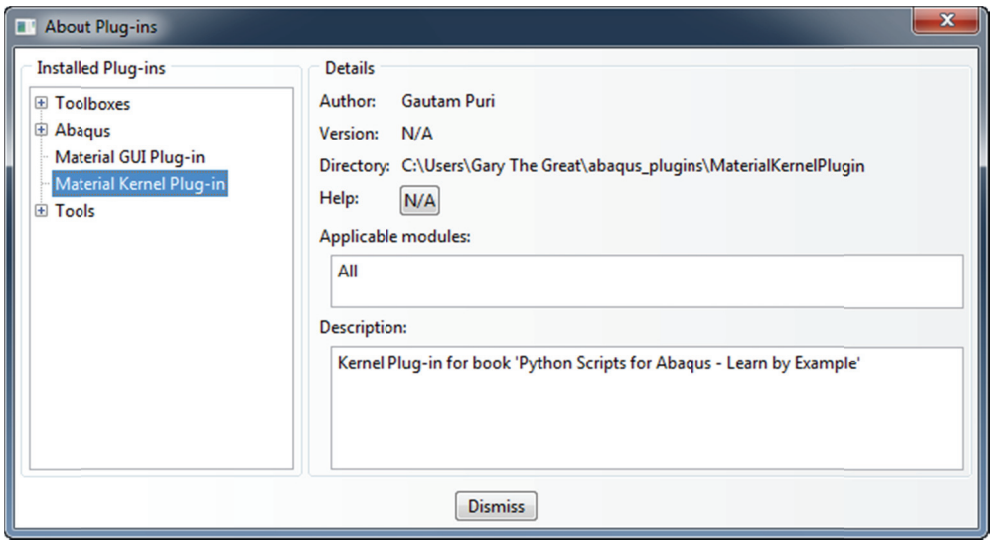
```
# ******************************************************************************
# Material Kernel Plug-in
# This script sends commands to the kernel to create the materials
# ******************************************************************************

from abaqus import *
from abaqusConstants import *

def createMaterials():

    mdb.models['Model-1'].Material('Titanium')
```

```
    mdb.models['Model-1'].materials['Titanium'].Density(table=((4500, ), ))
    mdb.models['Model-1'].materials['Titanium'].Elastic(table=((200E9, 0.3), ))

    mdb.models['Model-1'].Material('AISI 1005 Steel')
    mdb.models['Model-1'].materials['AISI 1005 Steel'].Density(table=((7872, ), ))
    mdb.models['Model-1'].materials['AISI 1005 Steel'].Elastic(table=((200E9, 0.29),
))

    mdb.models['Model-1'].Material('Gold')
    mdb.models['Model-1'].materials['Gold'].Density(table=((19320, ), ))
    mdb.models['Model-1'].materials['Gold'].Elastic(table=((77.2E9, 0.42), ))
```

We now create the plug-in. Here are the contents of 'materialkernel_plugin.py'

```
# ******************************************************************************
# Material Kernel Plug-in
# This script registers the material kernel plug-in
# ******************************************************************************

                                (Removed from Preview)
```

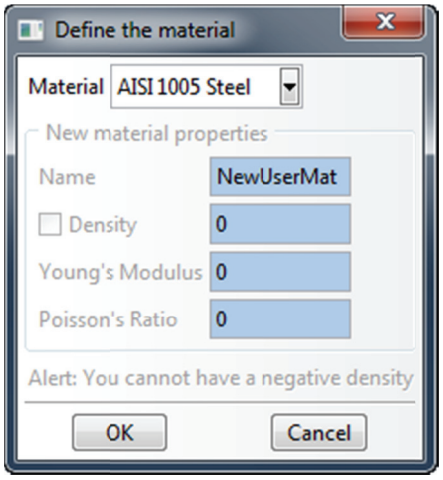
(Contents removed from preview)



This is all it takes to turn your kernel script into a functional kernel plug-in.

22.3.2 GUI Plug-in Example

We will reuse the material selection dialog box we created for the beam frame custom application in the previous chapter. This time it will appear as a standalone add-on rather than part of a full-blown custom application.



We reuse most of the code. **materialGuiDB.py** defines the dialog box, **materialGuiForm.py** defines the form mode that launches the dialog box, and **materialscript.py** is the associated kernel script.

The contents of **materialGuiDB.py** are the same as **step1DB.py** from the previous chapter.

```
from abaqusGui import *

# Class definition

class Step1DB(AFXDataDialog):

    [
                        ...
                        ...

    ] = range(AFXToolsetGui.ID_LAST, AFXToolsetGui.ID_LAST+4)

    #-------------------------------------------------------------------------
    def __init__(self, form):

                        ...
                        ...

    def onNegativeDensity(self, sender, sel, ptr):

                        ...
                        ...

    def onDensity(self, sender, sel, ptr):

                        ...
                        ...

    def onNewMaterialComboSelection(self, sender, sel, ptr):

                        ...
                        ...

    def onExistingMaterialComboSelection(self, sender, sel, ptr):
                        ...
                        ...

    #-------------------------------------------------------------------------
    def show(self):
                        ...
                        ...
```

```
    #-----------------------------------------------------------------------
    def hide(self):
                        ...
                        ...
```

The contents of **materialGuiForm.py** are the same as **step1Form.py** from the previous chapter.

```
from abaqusGui import *
import step1DB

# Class definition

class Step1Form(AFXForm):

    #-----------------------------------------------------------------------
    def __init__(self, owner):
                        ...
                        ...
    #-----------------------------------------------------------------------
    def getFirstDialog(self):
                        ...
                        ...
    #-----------------------------------------------------------------------
    def activate(self):
                        ...
                        ...
    #-----------------------------------------------------------------------
    def issueCommands(self):
                        ...
                        ...
```

As for **materialscript.py**, it is similar to the corresponding function from **beamKernel.py** of the previous chapter.

```
# ***************************************************************************
# Material GUI Plug-in
# This script sends commands to the kernel to create the material
# ***************************************************************************


                                (Removed from Preview)
```

Here is the script that actually creates the plug-in. It is **materialGui_plugin.py**.

```
# ****************************************************************************
                              # Material GUI Plug-in
                  # This script registers the material GUI plug-in
# ****************************************************************************

                              (Removed from Preview)
```

(Contents removed from preview)

22.4 Summary

Registering a plug-in is quite easy; you use the **registerKernelMenuButton()** and **registerGuiMenuButton()** methods depending on whether you are registering a kernel plug-in or a GUI plug-in. The real work goes into creating the kernel or GUI scripts that make up the plug-in. Once you have those, it's easy to package them into a plug-in for future use.