A-SV Interface Documentation

Release 2.0

bp, st & wy

Sep 19, 2017

CONTENTS:

CHAPTER

ONE

ABAQUS SCRIPTING USER'S GUIDE

1.1 5 Using Python and the Abaqus Scripting Interface

1.1.1 5.6 Extending the Abaqus Scripting Interface

You can extend the functionality of the Abaqus Scripting Interface by writing your own modules that contain classes and functions to accomplish tasks that are not directly available in Abaqus. For example, you can write a function to print the names of all materials that have a density specified, or you can write a function that creates a contour plot using a custom set of contour plot options. Creating functions and modules in Python is described in "Creating functions," Section 4.6.1, and "Functions and modules," Section 4.6.5.

This section describes how you can extend the functionality of the Abaqus Scripting Interface. The following topics are covered:

- "Storing custom data in the model database or in other objects," Section 5.6.1
- "Interaction with the GUI," Section 5.6.2
- "CommandRegister class," Section 5.6.3
- "Repositories," Section 5.6.4
- "Repository methods," Section 5.6.5
- "RepositorySupport," Section 5.6.6
- "Registered dictionaries," Section 5.6.7
- "Registered lists," Section 5.6.8
- "Registered tuples," Section 5.6.9
- "Session data," Section 5.6.10
- "Saving application data in a model database," Section 5.6.11
- "Checking a model database when it is opened," Section 5.6.12

5.6.1 Storing custom data in the model database or in other objects

If you extend the kernel functionality by writing your own classes and functions, you may want to store data required by those classes or functions in the Abaqus/CAE model database so the data are available the next time you open the database. To store custom kernel data in the Abaqus/CAE model database, you must make use of the `customKernel` module. The `customKernel` module augments the mdb object with a member called *customData*. When you save a model database, Abaqus/CAE also saves any data created below the customData object.

For example,

```
import customKernel
mdb = Mdb()
mdb.customData.myString = 'The width is '
mdb.customData.myNumber = 58
mdb.saveAs('custom-test.cae')
mdb.close()
```

If you start a new session and open the model database, `custom-test.cae`, you can refer to the variables that you saved. For example,

```
>>> import customKernel
mdb = openMdb('custom-test.cae')
>>> print mdb.customData.myString, mdb.customData.myNumber
The width is 58
```

You can store almost any type of Python object under `mdb.customData`; for example, strings, numbers, and Python classes. However, there are some restrictions; for example, you cannot store file objects. These restrictions are due to the fact that the Abaqus/CAE infrastructure uses Python's `pickle` module to store the *customData* object in the model database. The `pickle` module allows the Python programmer to write a data structure to a file and then recreate that data structure when reading from the file. For details on the restrictions imposed by the `pickle` module, see the official Python web site (www.python.org).

If your code creates a custom class and stores an instance of the class in the model database, the custom module that defined that custom class must be available for Python to unpickle the data when the database is subsequently opened. Consequently, if a user saves custom data to a model database and then passes that model database to another user, the other user must also have access to the custom modules that produced the custom data. Otherwise, they will not be able to load the custom data into their Abaqus/CAE session.

Abaqus/CAE does not keep track of changes made to the *customData* object. As a result, when the user quits a session, Abaqus/CAE will not prompt them to save their changes if they changed only objects under *customData*.

5.6.2 Interaction with the GUI

In addition to providing a persistence mechanism, the `customKernel` module contains classes that provide the following capabilities:

- Querying custom kernel data values from the GUI. From a GUI script you can access some attribute of your custom kernel object, just as you would from the kernel. For example,

  ```
  print mdb.customData.myObject.name
  ```

- Notification to the GUI when custom kernel data change. For example, you can have a manager dialog box that lists the objects in a repository. When the contents of the repository change, you can be notified and take the appropriate action to update the list of objects in the manager dialog box.

To make use of these features, you must derive your custom kernel objects from the classes listed in the following sections. For more details on GUI customization, see the Abaqus GUI Toolkit Reference Guide.

5.6.3 CommandRegister class

You can use the CommandRegister class to derive a general class that can be queried from the GUI. In addition, the class can notify the GUI when its contents change. For example,

```
class Block(CommandRegister):
  def __init__(self, name, ...):
```

```
    CommandRegister.__init__(self)
    ...
```

If a query is registered by the GUI on an instance of this class, the GUI will be notified when a member of this instance is changed, added, or deleted, For more details on registering queries, see the Abaqus GUI Toolkit Reference Guide.

If your object is to be stored in a repository (see below), the first argument to the constructor must be a string representing the name of the object. That string will automatically be assigned by the infrastructure to a member called *name*.

5.6.4 Repositories

Repositories are containers that hold objects that are keyed by strings. It may be convenient to store your custom kernel objects in repositories, in the same way that Abaqus/CAE part objects are stored in the `Parts` repository.

The customData object is an instance of a `RepositorySupport` class, which provides a `Repository` method that allows you to create a repository as an attribute of the instance. For more information, see “RepositorySupport,” Section 5.6.6. The arguments to the `Repository` method are the name of the repository and a constructor or a sequence of constructors. Those constructors must have *name* as their first argument, and the infrastructure will automatically assign that value to a member called *name*. Instances of these constructors will be stored in the repository. For more information, see “Repository object,” Section 53.3 of the Abaqus Scripting Reference Guide.

Since repositories are designed to notify the GUI when their contents change, the objects placed inside them should be derived from either `CommandRegister` or `RepositorySupport` to extend this capability to its fullest.

The Abaqus Scripting Interface uses the following conventions:

- The name of a repository is a plural noun with all lowercase letters.
- A constructor is a capitalized noun (or a combination of capitalized nouns and adjectives).
- The first argument to the constructor must be *name*.

For example, the `Part` constructor creates a part object and stores it in the `parts` repository. You can access the part object from the repository using the same name argument that you passed in with the `Part` constructor. In some cases, more than one constructor can create instances that are stored in the same repository. For example, the `HomogeneousSolidSection` and the `HomogeneousShellSection` constructors both create section objects that are stored in the `sections` repository. For more information, see “Abstract base type,” Section 6.1.5. For example, the following script creates a `blocks` repository, and the `Block` constructor creates a block object in the `blocks` repository:

```
from customKernel import CommandRegister
class Block(CommandRegister):
  def __init__(self, name):
    CommandRegister.__init__(self)

mdb.customData.Repository('blocks', Block)
block = mdb.customData.Block(name='Block-1')
print mdb.customData.blocks['Block-1'].name Block-1
```

5.6.5 Repository methods

Repositories have several useful methods for querying their contents, as shown in the following table:

Method	Description
keys()	Returns a list of the keys in the repository.
has_key()	Returns 1 if the key is found in the repository; otherwise, returns 0.
values()	Returns a list of the objects in the repository.
items()	Returns a list of key, value pairs in the repository.
changeKey(fromName, toName)	Changes the name of a key in the repository. This method will also change the name attribute of the instance in the repository.

The following script illustrates some of these methods:

```
from customKernel
import CommandRegister
class Block(CommandRegister):
  def __init__(self, name):
    CommandRegister.__init__(self)

mdb.customData.Repository('blocks', Block)
mdb.customData.Block(name='Block-1')
mdb.customData.Block(name='Block-2')
print 'The original repository keys are: ',
  mdb.customData.blocks.keys()
print mdb.customData.blocks.has_key('Block-2')
print mdb.customData.blocks.has_key('Block-3')
mdb.customData.blocks.changeKey('Block-1', 'Block-11')
print 'The modified repository keys are: ',
  mdb.customData.blocks.keys()
print 'The name member is ',
  mdb.customData.blocks['Block-11'].name
print 'The repository size is', len(mdb.customData.blocks)
```

The resulting output is

```
The original repository keys are ['Block-1', 'Block-2']
1
0
The modified repository keys are ['Block-11', 'Block-2']
The name member is Block-11
The repository size is 2
```

5.6.6 RepositorySupport

You can use the `RepositorySupport` class to derive a class that can contain one or more repositories. However, if you do not intend to create a repository as an attribute of your class, you should derive your class from `CommandRegister`, not from `RepositorySupport`.

Using the `RepositorySupport` class allows you to create a hierarchy of repositories; for example, in the Abaqus Scripting Interface the `parts` repository is a child of the `models` repository. The first argument passed into your constructor is stored as *name*; it is created automatically by the infrastructure. To create a hierarchy of repositories, derive your class from `RepositorySupport` and use its `Repository` method to create child repositories as shown below. The `Repository` method is described in "Repositories," Section 5.6.4.

```
from abaqus import *
from customKernel import CommandRegister, RepositorySupport
class Block(CommandRegister):
  def __init__(self, name):
    CommandRegister.__init__(self)

class Model(RepositorySupport):
  def __init__(self, name):
    RepositorySupport.__init__(self)
      self.Repository('blocks', Block)

mdb.customData.Repository('models', Model)
mdb.customData.Model('Model-1')
mdb.customData.models['Model-1'].Block('Block-1')
```

The path to the object being created can be found by calling `repr(self)` in the constructor of your object.

5.6.7 Registered dictionaries

You use the `RegisteredDictionary` class to create a dictionary that can be queried from the GUI. In addition, the infrastructure can notify the GUI when the contents of the dictionary change. The key of a registered dictionary must be either a String or an Int. The values associated with a key must all be of the same type—all integers or all strings, for example—to prevent errors when accessing them from the GUI. The `RegisteredDictionary` class has the same methods as a Python dictionary. In addition, the `RegisteredDictionary` class has a `changeKey` method that you use to rename a key in the dictionary. For example,

```
from customKernel import RegisteredDictionary
mdb.customData.myDictionary = RegisteredDictionary()
mdb.customData.myDictionary['Key-1'] = 1
mdb.customData.myDictionary.changeKey('Key-1', 'Key-2')
```

5.6.8 Registered lists

You use the `RegisteredList` class to create a list that can be queried from the GUI. In addition, the infrastructure can notify the GUI when the contents of the list change. The values in the list must all be of the same type—all integers or all strings, for example—to prevent errors when accessing them from the GUI. The values must all be of the same type; for example, all integers or all strings. The `RegisteredList` has the same methods as a Python list. For example, appending `Item-1` to the list in the following statements causes the infrastructure to notify the GUI that the contents of the list have changed:

```
from customKernel import RegisteredList
mdb.customData.myList = RegisteredList()
mdb.customData.myList.append('Item-1')
```

5.6.9 Registered tuples

You use the `RegisteredTuple` class to create a tuple that can be queried from the GUI. In addition, the infrastructure can notify the GUI when the contents of any of the members of the tuple change. The members in the tuple must derive from the `CommandRegister` class, and the values in the tuple must all be of the same type; for example, all integers or all strings. For example,

```
from abaqus import *
from customKernel import CommandRegister, RegisteredTuple
class Block(CommandRegister):
  def __init__(self, name):
    CommandRegister.__init__(self)

```

```
mdb.customData.Repository('blocks', Block)
block1 = mdb.customData.Block(name='Block-1')
block2 = mdb.customData.Block(name='Block-2')
tuple = (block1, block2)
mdb.customData.myTuple = RegisteredTuple(tuple)
```

5.6.10 Session data

The `customKernel` module also provides a session.customData object that allows you to store data on the session object and query it from the GUI. Data stored on the session object persist only for the current Abaqus/CAE session. When you close the Abaqus/CAE session, Abaqus does not store any of the data below `session.customData` on the model database. As a result, these data will be lost, and you will not be able to retrieve these data when you start a new session and open the model database. The session object is useful for maintaining data relevant to the current session only, such as the current model or output database.

The same methods and classes that are available for `mdb.customData` are available for `session.customData`.

5.6.11 Saving application data in a model database

If you have custom kernel scripts that store data in a model database, you may want to store information about your application in the same model database. When the model database is opened subsequently, you can access this information and decide how to proceed. For example, you can store version information and check if you need to upgrade your data in the model database.

You use the appData object to store custom application-related data in the model database. The appData object is an instance of an AbaqusAppData class. You can add any attributes to the appData object that are necessary to track information about your custom application. The following example illustrates how you can store the version number of your application on the appData object:

```
import customKernel
myAppData = customKernel.AbaqusAppData()
myAppData.majorVersion = 1
myAppData.minorVersion = 2
myAppData.updateVersion = 3
```

You use the `setAppData` method to install an appData object as session.customData.appData and to associate it with your application name. For example:

```
myAppName = ‘My App’
customKernel.setAppData(myAppName, myAppData)
```

You can call the `setAppData` method only once per application name, which prevents unauthorized changes to the method. However, the `setAppData` method may be called multiple times using different application names to allow more than one application to register with the same model database. When the user saves a model database, Abaqus copies the session.customData.appData object to the mdb.customData.appData object.

5.6.12 Checking a model database when it is opened

If you have custom kernel scripts that use custom data in a model database, you may want your application to verify some of the contents of a model database before it is fully opened. For example, you may want to check the database to see if you need to upgrade the data that is stored in it. In addition, you may need to initialize a new model database with your custom data. Two methods are provided for verifying and initializing a model database: `verifyMdb` and `initializeMdb`.

Verifying a model database

The `verifyMdb` method is used to verify the partial contents of a model database when it is opened. You must write the `verifyMdb` method and install it using the `setVerifyMdb` method. You can call the `setVerifyMdb` method only once per application name, which prevents unauthorized changes to the method. However, the `setVerifyMdb` method may be called multiple times using different application names to allow more than one application to register with the same model database.

When Abaqus opens a model database, its first action is to load only the mdb.customData.appData object and pass that object to each `verifyMdb` method registered in the session. If the model database has no appData, then Abaqus passes `None` to each `verifyMdb` method. Inside your `verifyMdb` method you can query the appData object to determine if you need to take any action, such as upgrading your data.

Initializing a model database

If a script creates a new model database, you can initialize the model database with your custom objects using the `initializeMdb` method. Abaqus calls each `initializeMdb` method registered with the session whenever a new model database is created. You must write the `initializeMdb` method and install it using the `setInitializeMdb` method. You can call the `setInitializeMdb` method only once per application name, which prevents unauthorized changes to the method. However, the `setInitializeMdb` method may be called multiple times using different application names to allow more than one application to register with the same model database.

Kernel initialization scripts specified by the **startup** command line option are executed by Abaqus/CAE after it has finished its initialization process. By that time, a new model database or a database specified on the command line using the **database** option has already been opened. A utility method called `processInitialMdb` has been created to automatically process the initial model database for you. If the initial model database does not have any *customData* or does not have *customData* for your particular application, your `initializeMdb` method will be called. If the initial model database has *customData* for your application, your `verifyMdb` method will be called.

The following example shows how you can use the `verifyMdb`, `intializeMdb`, and `processInitialMdb` methods. You should execute the example using the **startup** command line option when you start Abaqus/CAE. For more information, see "Abaqus/CAE execution," Section 3.2.6 of the Abaqus Analysis User's Guide.

```python
from abaqus import mdb, session
import customKernel
myAppName = 'My App'
myAppData = customKernel.AbaqusAppData()
myAppData.majorVersion = 1
myAppData.minorVersion = 1
myAppData.updateVersion = 1
customKernel.setAppData(myAppName, myAppData)
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def verifyMdb(mdbAppData):
  # If there is no appData, initialize the MDB.
  #
  if mdbAppData==None:
    initializeMdb()
    return
  # If my application is not in appData, initialize the MDB.
  #
  if not mdbAppData.has_key(myAppName):
    initializeMdb()
    return

  # Perform any checks on the appData or customData here

# Set the verifyMdb method for the application.
# setVerifyMdb may be called only once per application name.
```

```
#
customKernel.setVerifyMdb(myAppName, verifyMdb)

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def initializeMdb():
  # Initialize the MDB here


# Set the initializeMdb method for this application.
# setInitializeMdb may be called only once per application name.
#
customKernel.setInitializeMdb(myAppName, initializeMdb)

# This file is executed after Abaqus/CAE has started, so we need to
# process the initial MDB (either a new, empty MDB created by Abaqus/CAE,
# or a database opened via the -database command line argument).
#
customKernel.processInitialMdb(myAppName)
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search