Writing Python Scripts for Abaqus

L. Charleux 2009-2010

Contents

1 Introduction

Abaqus uses python scripting to communicate between modules like Abaqus/CAE, Abaqus/Viewer and various solvers. As a consequence, it is possible to use Python scripts to avoid using graphical interfaces (CAE and viewer). For exemple, one can use Abaqus build-in Python interpreters to:

- Create a CAE model using a Python script.
- Find information in an ODB result file.
- Submit jobs.

On one hand, if you run a small number of simulations are not really fond of programming, graphic interfaces are a good, maybe the best choice. On the other hand, if you run numerous simulations, parametric studies or any kind of problem you would like to automate, then Python scripting is the best solution for you.

It interresting to note that Python itself also provides a great amount of powerful tools to edit and modify text files like simulation input (INP) files which can be very usefull to run parametric studies. You just have to keep in mind that Python is an interpreted language unlike C/Fortran is an interpreted language. Where C and Fortran will be lead to fast executing programs, Python can be slow if you use to many for/if/while loops for exemple.

2 Starting with Python

While starting with abaqus Python interface, it is necessary to learn basics of Python language. For that purpose, websites allready exist and should be widely used:

- Python.org proposes a complete tutorial here:
  http://docs.python.org/tutorial/index.html
- *Think Python: How to Think Like a Computer Scientist* is available online here:
  http://www.greenteapress.com/thinkpython/html/index.html
- For 100 problems that seem strange to you, 99.9 have allready been solved and have a solution available on google so use it:
  Google

A last tutorial (paper version) is available for in our office if you need.

3 Abaqus Python and Python

What you do in Abaqus graphic interfaces is recorded in Python. For exemple, journal (JNL) files record all your actions in Abaqus/CAE as a Python script and all macros you build in Abaqus/(CAE, Viewer) are also Python scripts. You have to keep that in mind when you don't know how to perform an action, you can often do it with a macro and used the generated script to understand how you can do it with Python. There are several ways to run Abaqus built-in Python intepreters:

- `abaqus python` is the standard Python interpreter. We won't use it yet.
- `abaqus cae noGUI` and `abaqus viewer noGUI` are Python version of the two graphical interfaces. We will use them widely to perform actions that could have been performed by CAE and Viewer.

3.1 Getting started with the interpreter

In what follow, we use the Abaqus Viewer Python launched using:

```
abaqus viewer noGUI
```

If the interpreter is correctly lauched, you have some information about your abaqus version and a command line beginning with:

```
>>
```

Python commands can be typed directly, for exemple:

```
>> print 'Hello Larmaur !'
```

Will return:

```
Hello Larmaur !
```

You can also put all your commands in a text file called script named 'truc.py' for exemple and run it directly from a terminal using:

```
abaqus viewer noGUI=truc.py
```

For developping and debugging purpose, it's more confortable to run the interpreter in the interpreter and then run you script from it using:

```
>> execfile('./truc.py')
```

If you do so, you'll be able to use work on the various functions and variables you declare into the script after it runs.

3.2 Exploring ODB files

If your purpose is to explore ODB files, this section is for you.

3.2.1 Opening ODB files

To open an ODB file, you need the function `openODB` (which behaves like Python's `open`) available in the module `odbAccess`. You can load it using:

```
from odbAccess import openOdb
```

You need now to open you odb file using the following commands. Here the job is called `job` and so the ODB file should be named `job.odb`:

```
odbname = 'job'
path = './'
myodbpath = path + odbname + '.odb'
odb = openOdb(myodbpath)
```

Congratulations, your `odb` obejct is created and you can interact with it.

3.2.2 Digging in your ODB

Many objects are available in your ODB files:

- Everything you (or Abaqus/CAE) put into the INP file: parts geometry, materials, sections, step names, interaction property and anything else that appeared in this file.
- Everything you told Abaqus to save during the solving process: frames, field outputs, history outputs, time history and so on.
- Everything the solver saves like simulation status, the computer name...

Here you should realise how lazy you can get if you can get once your script is written. In order to find and proceed all this information, you should keep in mind some guidelines. Information has a tree structure in the `obd` object you created before. You can then get where you want in this tree if you know how to look what is available at your level. As a Python object, the odb looks similar to stacked lists, tuples and dictionnaries. As a consequence, a few Python commands will become your best buddies as an odb explorer:

- `print xxx` : this command should normaly be used to print a string but here you will use it as a way to know what is available at your level. Try it on your odb using `print odb` and you will see many subobjects availabe inside the odb. For exemple, you can find `'parts'`, `'steps'` and so on. You can then dig further using `print odb.steps` and you should find all the steps you chosed to run in your simulation. You can have a look in a step named `step1` using `print odb.steps['step1']`. If you want to have access to field outputs, this is a good way to proceed and you should normaly end using a command that looks like what follows to get the Von Mises stress :

  ```
  vonMises = odb.steps['step1'].frames[a].fieldOutputs['S'].values[b].mises
  ```

  Note that a is an integer referring to the frame and b is the number of the element. Note that in Python, lists items begin whith 0 and finish with $N-1$ where N is the number of elements in the list. On this point it is similar ot C programming. You can also use negative number to show that you read the list backwards. For exemple, if $a=-1$, you will work on the last frame of `step1` what is very usefull when you don't know how many frame there are.

- `len(xxx)`: this gives the length or the size of an object. It is very usefull when working with frames or elements because you do not allways know how many of the there is (or sometimes you forgot). For exemple, try:

  ```
  number = len(odb.steps['step1'].frames[a].fieldOutputs['S'].values)
  ```

  This command returns the number of objets in `values` that (as you allready guessed) is the number of elements. Then you can use `number` to run a `for` loop to use same command on all your elements in order to compute a plastic volume of anything else that should be integrated over a volume. For exemple, guess what the following commands do:

  ```
  volume = 0
  evol = odb.steps['step1'].frames[-1].fieldOutputs['EVOL']
  for element in number:  volume = volume + evol.values[element].data
  print 'volume = ' + volume
  ```

- `dir(xxx)`: this function returns the list of all the attributes and methods associated with an object (*i. e.* your obd object or some of its childrens).

3.2.3 Using Abaqus macros as learning tool

When you don't manage to find how to have your script perform a task, a solution can be to record the procedure using macros in Abaqus/viewer. Macros Manager can record a serie of actions performed by Abaqus Graphical User Interface (GUI) as a Python script that you can rerun afterwards. You can also use macros as a way to learn how the GUI performs the task you want to have your script do by editing `abaqusMacros.py` and pick some good scripting ideas.

3.3 Frequent requests

In this section, you should find some commands that can be helpful to perform various tasks.

3.3.1 Grabbing history outputs

Two ways are possible to get history outputs data:

1. The probably best way: you can find the history output named 'dataName' on the region 'region' during the step 'step' using:

   ```
   odb.steps['step'].historyRegions['region'].historyOutputs['dataName'].data
   ```

2. The not as good way: the cons of this way to proceed is that it requires the 'visualization' module which is not allways available on large servers and that it can allso be (very) tricky to find the true name of the data your are looking for. The pro is that you can use build functions available in the viewer like 'combine'. You can find the history output called `'name1'` during steps `step1` and `step2` at all frames using:

   ```
   from visualization import *
   xydata1 = session.XYDataFromHistory(name=name1, odb=odb,
   outputVariableName=xydata1, steps=(step1,step2), skipfrequency=0)
   ```

   With this command, you get a list of lists where the the first item is the frame and the second is the output (0 is time and 1 is `xydata1` at this time step) Note that you can do everything and more that what you can do in Abaqus viewer. You can use the commands available in the menu 'Operate on XYdatas' like `combine(a,b)` and so on. You can also write your own scripts to perform various tasks operating on XYdatas and export them in files using your own format or using abaqus RPT format.

3.3.2 Grabbing field outputs

Field outputs can be found in your `odb` object. For exemple, the Mises stress in step `step1` at frame 0 and element 10 can be found whith:

```
mises = odb.steps['step1'].frames[0].fieldOutputs['S'].values[10].mises
```

3.3.3 Grabbing materials

To know the material name associated with the section called `section1`, use:

```
mat = odb.sections['section1'].material
```

Then you can get information on the material with:

```
E= odb.materials[mat].elastic.table[0][0]
nu=odb.materials[mat].elastic.table[0][1]
sy=odb.materials[mat].plastic.table[0][0]
```

Where `E` is Young's modulus E, `nu` is Poisson's coefficient ν and `sy` is yield stress σ_y. Knowing what material was really used in your simulation can avoid big mistakes when running parametric simulations using numerous material properties. Abaqus tends not to write values equal to zero. For exemple, a zero Poisson coefficient may not be written in the table so make you script in a way that it finds out that the coefficient is zero instead of simply crashing the day you decide to use zero Poisson coefficient.

3.3.4 Grabbing parts

Information on parts can be found in:

```
odb.rootAssembly.instances
```

The nature this information strongly depends on the nature of your part (meshed, analytical surface, shell), each case should be explored differently. Working on parts is interresting for parametric studies as a fail-safe way to proceed. For exemple, you can have your script compute a length or an angle an write it in a report ot be sure that this simulation was running with proper parameters.

3.4 Exporting data

Now you know how to find a lot of data, you may want to save it somewhere. Many possibilities can be used depending on what you want. The main ones are:

1. Use text files: you can simply write your data in text files. It is a good idea if humans are supposed to read your data. It can not be the best solution if you save a large amount of data with a complex structure (not only a table) because you will spend time formating the output and time rereading the text file with another program. It will get worse when you decide to add

data to you outputs because you will have to updat all programs that should read the produced files. One can also argue that massive data like field outputs are not meant to be read by humans anyway.

2. Use serialization modules like `pickle` (or it's C fast clone `cpickle`). Such modules allow you to dump structured data like embedded lists, dictionnaries and tuples without thinking about the formating and reload it in another Python interpreter with the exact same structure using the `pickle.load` function. This method is fast and reliable but can be tricky when you want to read your data (pickle files are not meant to be read by humans) or to share data with non Python programs.