Development in code_aster
Creating a new macro-command

Code_Aster, Salome-Meca course material
GNU FDL licence (http://www.gnu.org/copyleft/fdl.html)
What is a macro-command?

Looks like a standard Code_Aster command.

\[
\text{acce2} = \text{CALC\_FONCTION(FFT=_F(FONCTION=acce, SYME='OUI'))}
\]

Has its own syntax description.

Usually calls some other commands.

May be very closed to usual command files

Written in Python but more or less « pythonized ».

Returns 0, 1 or N Code_Aster results.

Example of official macro-commands:

- ASSEMBLAGE: mainly a sequence of commands
- CALC_MODES: high level switcher between some more basic commands
- CALC_TABLE: works on Python objects converted from Jeveux objects
- CALC_EUROPLEXUS: calls an external program and build a Code_Aster object
Arguments of the macro-command

Arguments passing:

```python
def calc_function_ops(self, NOM_PARA, NOM_RESU, **args):
    """description""

    self is the macro-command objects.
    All existing keywords at the first level can be passed as arguments. You must check existence of those under blocks! Unnamed arguments are available with args[keyword].
```

Use the keywords value:

- High-level simple keyword: just use it as a usual variable
  - NOM_PARA contains a string, for example: ‘INST’
  - Simple keywords are list if max=‘**’ even if there is only one item.
- Python Keyword arguments (syntax **args)
  - args[‘INFO’] contains an integer: 1 or 2
- Loop on factor keyword occurrences (COMB can/should be repeated):
  ```
  for mcf in args[‘COMB’]:
      func = mcf[‘FONCTION’]
  # mcf[‘FONCTION’] is a fonction_sdaster
  Factor keywords always contains a list even if max=1.
  Simple keywords under each occurrence of COMB are available with mcf[keyword]
  ```

---

3 - Code_Aster and Salome-Meca course material

GNU FDL Licence
Under the hood of the macro-command

Always starts with

```python
self.set_icmd(1)
```

*Used by the supervisor for the numbering of commands: just copy/paste and forget it!*

**Import of Code_Aster commands:**

```python
from Cata.cata import LIRE_MAILLAGE, _F
```

**Import of utilities:**

```python
from Utilitai.Utmess import UTMESS
UTMESS('A', 'MESSAGE_NN', valk=..., vali=..., valr=...)
```

**Internal Code_Aster objects:**

The results of Code_Aster commands must not conflict with the user commands file.

```python
__mesh = LIRE_MAILLAGE()
```

*Beginning with 2 underscores, the result is temporary and will be automatically deleted at the end of the macro-command. It will be named ‘.99999xx’.*

```python
_mesh = LIRE_MAILLAGE()
```

*Beginning with one underscore, the result is hidden to the user, not deleted at the end, should be referenced by another object (model -> mesh for example). It will be named ‘_99999xx’.*

**Use two underscores as often you can (but only for commands results for readability)!**
Result(s) of the macro-command

Back to `sd_prod` method for more than one results:
- Returns the type of the main result
- Types the other results in its body (keywords declared with `typ=CO`)

Example:
```python
def more_complex_sdprod(self, MODELE, **kwargs):
    self.type_sdprod(MODELE, modele_sdaster)
    return maillage_sdaster
```

Usage:
```python
resmesh = MYCMD(MODELE=CO('resmodel'), …)
```

Link between the result and the local variable

For the main result:
```python
self.DeclareOut('mesh', self.sd)
mesh = LIRE_MAILLAGE(...)  # Only that when the macro returns one result.
```

Other results:
```python
self.DeclareOut('model', MODELE)
model = AFFE_MODELE(...)  # Only that when the macro returns one result.
```
Exercise

Create a macro-command, called CONV_MAIL_MED, that:

- reads a mesh file in the ASTER format (.mail),
- writes this mesh onto a MED file using a given logical unit number,
- reads this MED file in order to return a `maillage_sdaster` object.

**Inputs:**

- **UNITE_IN**: Logical number of a mesh file, compulsory.
- **FORMAT_IN**: Format of this mesh file, ‘ASTER’ by default, no other value for the moment
- **UNITE_OUT**: Logical number of the MED file, same default as in IMPR_RESU
  
  *This number is an input, the file is an output.*
- **INFO**: Verbosity flag that will be passed to subcommands. Optional, 1 or 2, default is 1.

**Output**

- The mesh (maillage_sdaster object).

**Improvements**

- Accept more formats in **FORMAT_IN**: ‘GIBI’, ‘GMSH’.

```
cd $HOME/dev/codeaster/src
hg pull -r a8b1cf codeaster_push
hg update -C a8b1cf
```
Use case

DEBUT()

master = CONV_MAIL_MED(UNITE_IN=20,
    INFO=2)

mgmsh = CONV_MAIL_MED(UNITE_IN=21,
    FORMAT_IN='GMSH',
    UNITE_OUT=81,
    INFO=2)

mgibi = CONV_MAIL_MED(UNITE_IN=22,
    FORMAT_IN='GIBI',
    UNITE_OUT=82,
    INFO=2)

FIN()

A testcase must really check the command using TEST_xxx operators.
CONV_MAIL_MED = MACRO(
    nom="CONV_MAIL_MED",
    op=OPS('Macro.conv_mail_med_ops.conv_mail_med'),
    sd_prod= ... ,
    fr=tr("Conversion d'un maillage vers le format Med"),
    reentrant= ... ,
    UNITE_IN= ... ,
    FORMAT_IN= ... ,
    UNITE_OUT= ... ,
    INFO= ... ,
)

from code_aster.Cata.Syntax import _F
from code_aster.Cata.Commands import PRE_GIBI, PRE_GMSH, LIRE_MAILLAGE,
    IMPR_RESU
from Utilitai.Utmess import UTMESS

def conv_mail_med(self, UNITE_IN, FORMAT_IN, UNITE_OUT, **kwargs):
    """Convert a mesh to the Med format""
    self.set_icmd(1)

    # print the message MESH_1
    UTMESS( ... )
    # declare the output
    ...
    # convert the mesh file if FORMAT_IN != 'ASTER'
    ...
    # read the input mesh file
    ...
    # write the med file
    ...
cata_msg = {
    1 : _(u"""
    Conversion du maillage au format %(k1)s, sur l'unité logique %(i1)d,
    au format MED dans le fichier d'unité logique %(i2)d.
    """"),
}

Expects two integers and one string.

The message identifier is built from the filename (uppercases) and the message number:
Message #1 in mesh.py => MESH_1
Debugging a macro-command

Prepare of the execution directory with

```bash
waf test -n tpdvp04a -exectool=env
```

Open the output, and copy/paste the instructions:

```bash
cd /tmp/user-hostname-interactif.11144
[There may be several environment files]
. $HOME/dev/codeaster/install/std/share/aster/profile.sh
```

To start execution in the Python debugger you could type :

```bash
cp fort.1.1 fort.1
`/install/std/bin/aster /usr/lib/python2.7/pdb.py
`/install/std/lib/aster/Execution/E_SUPERV.py -commandes fort.1 -num_job 11144 -mode interactif -rep_outils /opt/aster/outils -rep_mat .../install/std/share/aster/materiau - rep_dex .../install/std/share/aster/datg -tpmax 60.0 -memjeveux 126.5
(Pdb) break /home/mc/dev/codeaster/install/std/lib/aster/Macro/conv_mail_med_ops.py:29
Breakpoint 1 at /home/mc/dev/codeaster/install/std/lib/aster/Macro/conv_mail_med_ops.py:29
(Pdb) continue
...
(Pdb) print UNITE_IN
20
(Pdb) next
...
(Pdb) list
...
```

+ Trace subcommands calls with:
  DEBUT(IMPR_MACRO='OUI')
Coding tips

• Use small functions
• Take benefit of the Object-Oriented Programming
  Example of CALC_FONCTION:
    • Each factor keyword calculates a function using different algorithms but the reading of inputs and the build of the output function are common.
    • Each factor keyword is a different subclass of an abstract one.
    • Adding a new calculation is easy.
• Document all functions, classes... you will save this time later!
• The source code must respect the PEP8 conventions
End of presentation

Is something missing or unclear in this document? Or feeling happy to have read such a clear tutorial?

Please, we welcome any feedbacks about Code_Aster training materials. Do not hesitate to share with us your comments on the Code_Aster forum dedicated thread.