To introduce a new macro-order

Summary:

This document describes how to define and use the macro-orders in python.

Contents

1 Introduction

This document describes the use and the development of the macro-orders in Python for Code_Aster. These macro-orders can be restored in the code and visible of the user like orders except for whole. But they being able to also be placed in the command file itself without the user has to touch neither with the achievable one, nor with the catalogue of orders. Moreover, they have the advantage of being written “naturally” in the process control language: the body of an macro-order is similar to an ordinary command file which would generate the same sequence of orders.

The developer will may find it beneficial great to read the programming of existing macro-orders, under `Macro bibpyt/` in the repertoire of installation of *Code_Aster* .

2 What macro?

Macro is an order which gathers the execution of several subcommands. It is usable in a command file like any other order and has its own catalogue, defining its syntax. Several types of macro are possible:

- macros specific to the supervisor, implemented in Python and FORTRAN: for example `FORMULA` , `INCLUDE` , `BEGINNING` ...
- macros developers implemented in Python.

This document treats definition and use of the macros in Python.

3 To use the macros

To use the macros in Python is simple. Compared to simple orders like `OPER` or of `PROC` , the only difference relates to the concepts produced by the macro one. A simple order, of type `OPER` , only one produced concept has which one will find on the left of the sign “=”, as follows:

```
concept = ORDER (word-key-simple-or-factors)
```

A simple ordering of type `PROC` no produced concept has and is written:

```
ORDER (word-key-simple-or-factors)
```

An macro-order, of type `MACRO` , can have several produced concepts. One (but it is not obligatory) which one will find on the left of the sign =, as for one `OPER` , others like arguments of the simple keywords or factors. One will present the instructions for a simple keyword. It extends easily to the keywords factors. Certain keywords are likely to produce concepts. To ask a macro order to produce this concept, the user will write on the right of sign = following the name of the keyword, `CO` (‘ `nom_concept` ’) , as in the example which follows:

```
ASSEMBLY (NUMEDDL=CO ('num'))
```

This causes to create a produced concept of name `num` at exit of the order `ASSEMBLY` . Its type will be given according to the conditions of call of the order. `CO` is a reserved name which makes it possible to create named produced concepts, not typified, prior to the call of the order. It is the order which will allot the good type to this concept.

4 To define an macro-order in Python

It is necessary to define:

- the catalogue itself of the keywords composing the macro one,
- method of typing of the structures of produced data,
- the method Python defining the body of macro: “basic” orders produced by the macro one and their sequence.

The first two points are common with the writing of an ordinary order (except for a minor difference in the method of typing).

It is possible to restore all this in the catalogue of orders of the code and to thus make the macro-order visible of all. One can also preserve his development deprived with the advantage of not having to modify the parameters of execution or the achievable one while placing these three elements directly at the top of the command file, or in the important one (importation Python) since an agreed localization.

4.1 To write the catalogue of the macro-order

The catalogue of macro is similar to that of a simple order. The three differences are:

- an object is declared MACRO (and not PROC or OPER),
- the reserved keyword op an entirety (indicating does not contain the number of high level routine FORTRAN for OPER and PROC) but a name of method python,
- the produced concept necessarily single, is not declared on the left a sign “=”.

Produced concepts can be specified as arguments of a simple keyword. U N keyword simple who defines a concept to be produced is declared of type CO :

```
NUME_DDL = SIMP (statut='F', typ=CO)
```

The type of the turned over object will be defined in the function sd_prod (cf following paragraph).

The same keyword can play a double role: to take a concept in argument or to produce a new concept. For example, one would write:

```
NUME_DDL = SIMP (statut='F', typ=(nume_ddl_sdaster, CO))
```

This use is strongly disadvised for the future developments. It is advised to define a keyword for each function.

4.2 To define the type of the produced concepts

The definition of the type of the produced concepts of an macro-order is carried out in a way similar to that of a simple ordering of type OPER .

If the order produces only one concept which one will find on the left of the sign = as in:

```
has  =COMMANDE ()
```

one will proceed in the same way as for a simple order of type OPER .

If the macro-order can produce several concepts whose some in arguments of keywords, should be added some extra information. First of all, it is necessary absolutely to provide a function Python, named sd_prod , in the definition of macro. Then, the simple keywords containing the name user of the concept to be produced must be of type CO (reserved name).

For example:

```
def ma_macro_prod (coil, NUME_DDL, MATRIX, ** arguments):
     yew args.get ('__all'):
         return ([evol_noli], [nume_dll_sdaster], [Nun, matr_asse_depl_r])

    self.type_sdprod (NUME_DDL, nume_ddl_sdaster)
     yew STAMPS:
        self.type_sdprod (MATRIX, matr_asse_depl_r)
    return evol_noli
….
MA_MACRO =MACRO (sd_prod=ma_macro_prod,…
                 MATRIX  = SIMP (statut='F', typ=CO),
                 ….
                 NUME_DDL = SIMP (statut=' o', typ=numérique_ddl_sdaster),)
```

In this case, three concepts can be produced:

- in a classical way, a concept of the type `evol_noli` who will have been given by the user on the left of the sign "=".
- a concept of the type `matr_asse_depl_r` if the user inform the simple keyword `MATRIX`.
- a concept `nume_ddl` whose name is provided to the keyword `NUME_DDL`.

When a keyword can have in argument a concept to produce, the keyword must appear in the list of the arguments of the function `sd_prod` and the concept must be typified by using the method `type_sdprod` argument self-service who is the macro-order object.

NB : The argument `coil` is not present for one `OPER` or one `PROC` (it is the object `MACRO`).

As for the orders (cf [D5.01.01]), the function `sd_prod` must turn over the whole of the possible types if `__all__=True` . As the macro-order can turn over several objects (3 in the example), if `__all__=True` , the function turns over a list of 3 lists of the possible types, a list for each result.
If a result is not systematically produced (here for `MATRIX`), the possible types are then `Nun` (nothing is produced) and true types.
S I the macro-order does not turn over a result on the left of the sign "=", the first list is worth then `[Nun]` (instead of `[evol_noli]` here).
In certain cases, macro-orders turn over an indefinite number of results (one by occurrence of a keyword factor). In this case, the corresponding list is not repeated.

5 To define the body of macro

5.1 Organization of the modules

In the case of macro deprived, the user organizes these modules Python as it wishes it.

On the other hand for an macro-order integrated into the source official, it is necessary to comply with certain rules.
The files are at least:

- `nom_macro.py` : the catalogue of the order
- `nom_macro_ops.py` : described body the macro one (by defining the principal method by `def nom_macro_ops (…)`)

Recommendations

- In `nom_macro.py` , the method should not be imported `nom_macro_ops` so that the catalogue is self-supporting (to be usable with uniquemnt contents of

code_aster/Catastrophes). One is satisfied to define “the address” towards the method, and specifying:

```
op=OPS ('Macro.nom_macro_ops.nom_macro_ops')
```

The method defining the body of the macro-order will be nom_macro_ops defined in the file (module) nom_macro_ops.py repertoire (package) Macro .

- The definition of the method nom_macro_ops must respect conventions standards of coding in Python. In particular, one should not make a function of 1000 lines: to cut out in elementary functions. Preferably, to isolate these elementary functions in another module which will be imported under the body of nom_macro_ops .

5.2 Transmission of the keywords the macro one with the method of construction (the body)

The body the macro one will be defined in a method of the object MACRO whose arguments are the arguments of the keywords. The first argument is the macro-order object, coil , the following is the keywords necessary to express the body of macro. Keywords optional or whose presence is conditioned by blocks will be omitted by the use of the argument ** arguments .

Only the high level keywords are transmitted: MCSIMP of first level, MCFACT. These keywords are then called upon very simply by their name.

Example of function body:

```
def ma_macro_ops (coil, UNITE_MAILLAGE, ** arguments):
    ..................
          _NOMLMA = LIRE_MAILLAGE (UNIT = UNITE_MAILLAGE)
    ..................
```

Here, UNITE_MAILLAGE the macro one, its contents is a MCSIMP (concept, list, G-string,… it does not matter) is affected with the MCSIMP UNIT order LIRE_MAILLAGE .

Case of a keyword factor:

```
def ma_macro_ops (coil, MATR_ASSE_GENE, ** arguments):
    ..................
          yew MATR_ASSE_GENE.get ('MATR_ASSE') is None:
    ..................
```

MATR_ASSE_GENE the macro one is a MCFACT, MATR_ASSE is one of its under-MCSIMP. A MCFACT is handled like a dictionary.

Trick: in this last example, one tests very simply the presence of MATR_ASSE : if the user did not inform a keyword (simple or factor), it is not in the dictionary and thus the method get () turn over Nun .

Notice

If a simple keyword accepts only one value (max=1 in its definition), then in the macro-order, this keyword turns over the well informed value. For example, INFORMATION turn over 1 or 2 (a single entirety).
*If a simple keyword is defined with max>1 or max=' ** ', then the turned over value will be always a tuple even if the user provided only one value. For example, GROUP_MA (often definite with max=' ** ') will turn over ('GM1', 'GM2') or ('GM1',) if there is only one value.*
For a keyword factor, one always turns over a list.

5.3 To call an order in the body the macro one

To call an order in the body the macro one, it is necessary to import its catalogue (to gather all together imports) :

```
from code_aster.Cata.Commands importation NUME_DDL
num = NUME_DDL (METHODE=…,…)
```

There is no obstacle so that the orders “girls” produced by the macro-order are they-even macro-orders.

If one definite a FORMULA in the body of an macro-order and that in the expression of this one one uses constants (for example has in VALE= ''' a*INST '''), it is necessary to declare the value explicitly before evaluating of it the formula:

```
self.update_const_context ({'has': 2. })
… then CALC_FONC_INTERP or other…
```

5.4 Naming of the concepts produced in the body by the macro one

The concepts produced in the body of macro are several kinds:

- concepts named and destroyed automatically at the end of the execution of the macro-order. One thus should not any more need some thereafter and it should in particular be taken care that the concept produced by the macro one does not refer there. To indicate that it is about a concept of this kind, it is enough to give him a name which starts with __(double underscore)

Example:

```
__a=CALC_MATR_ELEM (MODELE=MODELE)
```

Then, as one can read it in the file of messages, an automatic name of concept, preceded by a point is generated:

```
.9000005=CALC_MATR_ELEM (MODELE=MODELE)
```

Once left macro, the object corresponding in the name of concept .9000005 exist any more, neither within the space of names of the supervisor, nor in the base jeveux .

- concepts named automatically and preserved in the base jeveux at the end of the macro-order. To indicate that it is about a concept of this kind, it is enough to give him a name which starts with _ (simple underscore)

Example:

```
_a=CALC_MATR_ELEM (MODELE=MODELE)
```

Then, as one can read it in the file of messages, an automatic name of concept, preceded by a underscore is generated:

```
_9000005=CALC_MATR_ELEM (MODELE=MODELE)
```

Once left macro, the object corresponding in the name of concept _9000005 do not exist within the space of names of the supervisor, there is on the other hand present under this name in the base jeveux .

This kind of object answers the typical locations where the concept produced by the macro one "depends" on a concept upstream which will have to be always present: for example one model compared to a grid, one matr_asse compared to one nume_ddl .

- concepts intended to become produced concepts of macro. To indicate that it is about a concept of this kind, the method should be invited DeclareOut macro object coil with, like arguments, the name of the variable back from the order and the object resulting from the keywords of the macro-order.

Example: the local variable is associated mm with the keyword MATRIX the macro one:

```
self.DeclareOut ('mm', STAMPS)
mm=ASSE_MATRICE (.............)
```

- the concept of exit the macro one (necessarily single) is treated in a similar way by indicating the concept of exit self.sd .

```
self.DeclareOut ('mm', self.sd)
mm=ASSE_MATRICE (.............)
```

mm of exit the macro one will become the concept, it will bear the name given by the user in his command file (and not mm).

5.5 Recovery of the exceptions in macro

The operation of the particular and different supervisor being according to whether an order is carried out in the command set or an macro-order, the treatment in the event of exception also must is particular.
Let us see that on an example in the command set:

```
try:
. . resu = STAT_NON_LINE (...)
except NonConvergenceError ace exc:
. . /code/
```

In /code/, one has access to resu. Why? because as of the call to the order, resu is placed in jdc.g_context and that the contents of the command set are carried out in this same context.

If one needs to make this kind of recovery in macro, that does not function in this manner, it is more CoNform with code ordinary Python.
It is necessary to call on a method dedicated to that: get_last_concept (). This one makes it possible to recover the shell assd on the last concept produces in this macro.
One will make then:

```
try:
. . __resu = STAT_NON_LINE (...)
except NonConvergenceError ace exc:
. . resu = self.get_last_concept ()
. . /code/
```

__resu is the temporary concept created in the macro one (under for example the jeveux name . 9000027).

resu is an object assd who gives access the structure of data FORTRAN of name .9000027. In this case, resu. Liste_Vari_Acces () will function.

If the order emits an error 'F', an object nevertheless will be recovered assd but the structure of data FORTRAN will not exist because one will have seen the message "Destruction of the concept '.9000027'"(from where interest to recover only the typified exceptions, as in the example, if not it is very difficult to know in which state is the concept).

In the examples above, the variable exc the object derived contains from aster.error.
This object lays out of attribute making it possible to know which exception was raised and with which identifier of message.

5.6 To reach the concepts produced before the macro one

Generally, all the concepts necessary to operation the macro one must forward by its keywords.

In some typical cases, it is necessary to recover a concept which one knows only the name. For example, the name is stored in a structure of data as starter. To obtain the concept starting from a name, the method should be used get_concept object MACRO. Maybe in the body the macro one:

```
object = self.get_concept (name)
```

In the same order of idea, one can need the whole of the objects available (for example, STANLEY propose the choice among the already calculated concepts). For that, one recovers the context like this:

```
dictionary = self.get_contexte_courant ()
```

Method get_concept give access to the concepts recorded near the command set. A temporary concept created in an macro-order but which was hidden with the user (named for example _9000028), cannot be recovered directly. It is necessary to create a reference towards the structure of data FORTRAN while utuilisant get_concept_by_type. Example:

```
object = self.get_concept_by_type ('_9000028', maillage_sdaster)
```

NB: the use of the dictionary sds_dict is to be proscribed (for the two above mentioned tasks), it acts of an internal object to the supervisor.

5.7 To number the macro one

In postings of the file of messages, all the orders are numbered. To increment this meter, it is systematically necessary to invite the following method at the top of the body the macro one:

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

It is particularly important to make to this call before any order “girl” of the macro order because this method also initializes the total measurement of time CPU for the macro one.

5.8 To treat the errors

As for an order in FORTRAN, it is possible to detect errors of use in the body the macro one by the Utmess utility, identical in its operation to its homonym FORTRAN ([D6.04.01] - Utility of impression of messages). With this intention, it is necessary to import this method since the module of the utilities Python.

Example:

```
from Utilitai.Utmess importation UTMESS
…
UTMESS ('F', 'CHARGES_4', valk= " DX", vali= (2, 88))
```

The first argument indicates the nature of the error or alarm, the second specifies the identifier of the message in the catalogue of messages and the arguments following (vali, valr, valk) allow to provide integer, real variables or character strings to supplement the message.

5.9 Postings

As far as that is possible, the messages bound for the user will be displayed with UTMESS ('I',…) .
That makes it possible to define coherent messages and which could be translated automatically.
When it is not possible to use the catalogue of messages, it is possible to directly print text on the file MESSAGE or RESULT by employing the method poster module aster .
The use of the order print is strongly disadvised.

Example of use of aster.affiche :

```
importation aster
…
aster.affiche ('MESSAGE', chaine_de_caracteres)
```

The first argument is worth ' MESSAGE ' or ' RESULT ' according to the target file.

5.10 To identify the concepts produced by the macro one

In certain circumstances, it is necessary to determine if a concept is produced by macro itself or were produced by a preceding order. This is possible while testing if the concept in question is present or not in the list of the concepts produced by the macro one which is given by the attribute sdprods object macro coil.

Example:

```
yew nume_ddl in self.sdprods:
   # the concept nume_ddl is produced by the macro one
   # it is necessary to call order NUME_DDL
   lnume = 1
else:
   # the concept nume_ddl already exists.
   lnume=0
```

Attention, sdprods the produced concept turned over by the macro one does not contain which is in the attribute sd of coil .

5.11 Dynamic creation of orders: number of variable keywords, contextual contents

In certain cases, according to the value of the options, the same order will be called with various keywords or of the different arguments. To treat this situation and to generate the order dynamically, one builds a dictionary containing the keywords to be written which is then transmitted in argument of the order, preceded by the characters '**'. The dictionary "is then unfolded", the keys are the arguments (keywords), followed by the contents of the key, behind the sign "=".

This dictionary Python can be built as the examination of the options.

Example:

```
moscles= {}
moscles ['INFORMATION'] = 2

keywords ['CREA_GROUP_NO'] = []
for grma in GROUP_MA_BORD:
```

```
        keywords ['CREA_GROUP_NO']. suspends (_F (GROUP_MA = grma))

    _nomlma = DEFI_GROUP (  reuse = _nomlma
                                          GRID = _nomlma,
                                          ** keywords)
```

The dictionary `keywords` the definition contains of `INFORMATION` and a list of keywords factors `CREA_GROUP_NO` . For the example, the list is a simple recopy of `GROUP_MA_BORD` built by addition successive of an element.

5.12 Call to an external code

If one wishes to carry out in macro third code by the order `EXEC_LOGICIEL` or by using the module `Utilitai.System` , one must recover the way towards the external software. If it is about an officially supported software, its access path is recorded in a named file of the installation `external_programs.js`. One recovers it while making:

```
importation aster_core
miss3d = aster_core.get_option ('prog:miss3d')
EXEC_LOGICIEL (…, LOGICIEL=miss3d,  …)
```

5.13 Rules of programming

The compliance with rules of programming is necessary to improve the legibility and the coherence of the code. The absence of checking of these rules of programming leads to a great disparity of the code Python of *Code_Aster* what makes difficult the fast comprehension of this one.

It would imperatively be necessary to conform to the recommendations of the PEP 8 – Style guides for Python codes:
http://www.python.org/dev/peps/pep-0008/

Some rules simple to respect:

- Indentations: 4 spaces (and not 2 or 3), not of tabulations.
- Maximum width of 80 characters (up to 90-100, it is tolerable, plus that becomes really illegible).
- Only one instruction per line (not of instruction on the same line as `if/else`).
- Only one `importation` by line, never of '`importation *`'.
- Not to multiply the white lines of separation: two between the functions/classes with the more high level, one inside a class.
- A space enters the operators (+, -, *…), after the comma, afterwards ': ' the reading facilitates.
- The chain Doc. of the functions is obligatory to describe what they do.
- To in small letters name the variables, the classes with a capital letter with each word…

In particular for *Code_Aster* :

- In a glance, one must understand what does macro or a function: to cut out the code in elementary tasks of 50-100 lines each one. To separate in several files so that remains readable (<500 lines).
- Keywords being in capital letters, to use the tiny ones for the variables.
- The naming of the concepts is sufficiently complicated to take care to name them in small letters for good to distinguish them from the keywords. Except the produced concepts, they start with one or 2 '_'. Not to name ordinary variables Python while starting with a '_'.

Tools: `reindent.py` provides with Python allows réindenter of the code according to convention, `pylint` fact a diagnosis of the respect of conventions…

6 Consideration on the use of the macros in Python

6.1 Definition of macro except catalogue

Standard method to add the definition of macro in Python for an execution of *Code_Aster* is to add it in the catalogue of reference of the code.
However, in certain cases:

- macro personal,
- test during the development,

it can be practical to add the definition of macro apart from the catalogue. With this intention, it is enough to create a module Python containing the definition of macro by adding at the top of the module the importation of the variables of the catalogue.

Simplified example:

```
from code_aster.Cata.Syntax importation…
from code_aster.Cata.DataStructure importation…
def ma_macro_prod (coil,…):
    .....
def ma_macro_ops (coil,…):
….
MA_MACRO=MACRO (nom=' MA_MACRO',…)
```

Then with the use, the weather is enough in the command file to be the importation of macro previously definite.

Example of command file:

```
# the macro MA_MACRO is defined in the module ma_macro.py
from ma_macro importation MA_MACRO
a=MA_MACRO (...)
```

It is also possible to use the functionality of`INCLUDE` :

```
# the macro MA_MACRO is defined in file INCLUDE 45
INCLUDE (UNITE=45)
a=MA_MACRO (...)
```