To introduce a new macro-command

Summarized:

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

Contents

1 Introduction3.....
2 What macro? 3
3 To use the macros3.....
4 To define a macro-command in Python4.....
4.1 To write the catalog from..... macro-commande4
the 4.2 To define the type of the concepts produits4.....
5 To define the body of the macro5.....
5.1 Organization of the modules5.....
5.2 Transmission of the keywords the macro one in the method of construction (the body)..... 6
5.3 To invite a command in the body of the macro6.....
5.4 Naming of the product concepts in the body by the macro7.....
5.5 Recovery of the exceptions in a macro8.....
5.6 To reach the product concepts before the macro8.....
5.7 To number the macro9.....
5.8 To treat the erreurs9.....
the 5.9 affichages9.....
5.10 To identify the product concepts by the macro10.....
5.11 dynamic Creation of commands: many variable key words, contained contextuel10.....
5.12 Call to a code externe10.....
5.13 Rules of programmation11.....
6 Consideration on the use of the macros in Python11.....
6.1 Definition of macro except catalogue11.....

1 Introduction

This document describe the use and the development of the macro-commands in Python for Code_Aster. These macro-commands can be restored in the code and visible of the user like commands 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 executable one, nor with the catalog of commands. Moreover, they have the advantage of being written “naturally” in the process control language: the body of a macro-command is similar to an ordinary command file which would generate the same sequence of commands.

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

2 What macro?

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

- macros specific to the supervisor, implemented in Python and FORTRAN: for example `FORMULA` , `INCLUDE` , `debut` ...
- of the 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 commands like `OPER` or `PROC` , the only difference relates to the product concepts by the macro one. A simple command, of type `OPER` , has only one product concept which one will find on the left of the sign “=”, as follows:

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

a simple command of type `PROC` does not have any product concept and is written:

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

a macro-command, of MACRO `type` , can have several product concepts. One (but it is not compulsory) which one will find on the left of the sign =, as for `an OPER` , the others like arguments of the simple key words or factors. One will present the instructions for a simple key word. It extends easily to the key words factors. Certain key words are likely to produce concepts. To ask a macro command to produce this concept, the user will write on the right of sign = following the name of the key word, `CO` (“ `nom_concept` ”) , as in the example which follows:

```
ASSEMBLY (NUMEDDL=CO (“num”))
```

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

4 To define a macro-command in Python

It is necessary to define:

- the catalog itself of the key words composing the macro one,
- method of typing of produced data structures,
- Python method defining the body of macro: “basic” commands produced by the macro one and their sequence.

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

It is possible to restore all this in the catalog of commands of the code and to thus make the macro-command visible of all. One can also preserve his development deprived with the advantage of not having to modify the parameters of execution or the executable 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 catalog of the macro-command

the catalog of macro is similar to that of a simple command. The three differences are:

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

Product concepts can be specified as arguments of a simple key word. If a simple key word can accept a product concept like argument, it is necessary, besides the type, to specify `CO` in the tuple `typ` .

Simple example of key word accepting a product concept or an existing concept:

```
NUME_DDL  =SIMP (statut=' o', typ= (nume_ddl, CO))
```

Here, the key word accepts in argument an existing concept of nume_ddl `type` or a concept to be produced of a type which will be given by the command.

4.2 To define the type of the product concepts

the definition of the type of the product concepts of a macro-command is carried out in a way similar to that of a simple ordering of type `OPER` .

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

```
=COMMANDE  has ()
```

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

If the macro-command can produce several concepts whose some in arguments of key words, should be added some extra information. First of all, it is absolutely necessary to provide a function Python, named `sd_prod` , in the definition of macro. Then, the single-ended spanner keys 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, MATRICE, ** arguments):
    yew NUME_DDL.is_typco (): self.type
        _sdprod (NUME_DDL, nume_ddl_sdaster) self.type
    _sdprod (MATRICE, matr_asse_depl_r) return evol
    _noli…. MA_
MACRO
=MACRO (sd_prod=ma_macro_prod,… MATRICE
                = SIMP (statute  = ' o', typ=CO),…. NUMERICAL
                _DDL
                = SIMP (statut=' o', typ= (CO, nume_ddl_sdaster)),) In this
case,
```

three concepts can be produced: in a classical

- way, a concept of the evol_noli type `which` will have been given by the user on the left of the sign "=" a concept
- of the type matr_asse `_depl_r whose` name will have been provided by the user behind key word simple MATRICE `for`
- case NUME_DDL , `the user` has the choice between providing here an already existing `concept numeddl` or making it produce by the macro one, in which case it determines itself its naming as in case MATRICE . `Lorsqu`"

a key word can have in argument a concept to produce, the key word must appear in the list of the arguments of the function sd_prod `and the concept` must be typified by means of the method type_sdprod `of L" argument` self-service which is the macro-command object. NB: The argument

self-service `is not` present for an OPER or a PROC (it `is` the MACRO object). `To define`

5 the body of the macro Organization

5.1 of the moduli In the case of

macro deprived, the user organizes these moduli Python as it wishes it. On the other hand

for a macro-command integrated into the source official, it is necessary to comply with certain rules. The files
are at least: nom_macro.capy

- `: the catalog` of the command nom_macro_ops.py
- `: described body` macro (by defining the principal method by def nom_macro_ `ops ()…)` `Recommendations`

In nom_macro.capy

- `, one should not` import the method nom_macro_ops so that `the catalog` is self-supporting (to be usable in Eficas, without this one needing the Macro directory). 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-command will be nom_macro_ops defined `in the file` (modulus) nom_macro_ops.py of the directory `(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, isolate these elementary functions in another modulus which will be imported under the body of nom_macro_ops. Transmission `of the keywords`

5.2 the macro one with the method of construction (the body) the body the macro one will be

defined in a method of the MACRO object whose arguments are the arguments of the key words. The first argument is the macro-command object, coil, the following are `the key words` necessary to express the body of macro. The useless key words 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 (UNITE
          = UNITE_MAILLAGE) ................. Here, UNITE_MAILLAGE
   is
```

a MCSIMP `the macro one,` its contents (concept, list, string,… it does not matter) is affected with MCSIMP UNITE of the command LIRE_MAILLAGE . Case of `a key word factor`

: def ma_macro_ops (coil, MATR_ASSE_GENE

```
, ** arguments): ................. yew MATR_ASSE_GENE
   ["MATR_ASSE
          "] is None: ................. MATR_ASSE_GENE
   is a MCFACT
```

`the macro one,` 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 key word (simple or factor), it is worth by default Nun. Notice If a simple key word

accepts

only one value (max=1 in its definition) , `then` in the macro-command, this key word turns over the well informed value. For example, INFO turns over 1 or 2 (a single integer). If a simple key word 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 . To invite a command in*

5.3 the body the macro one to call a command

in the body the macro one, it is necessary to question the catalog by L" intermediate of the method get_cmd of L" object macro-command coil to obtain this command : NUME_DDL=self.get_cmd ("NUME_DDL

```
") num=NUMÉRIQUE_DDL (METHODE=…,…)
It is essential to use
```

the exact name of the command as name of the variable which will contain the return of the method get_cmd. Indeed this name `is used` to locate line text containing this name and thus to identify the name of the product concept (here num). There is no obstacle so that the commands “girls” produced by the macro-command are they-even macro-commands. If one definite a FORMULA in

the body of a macro-command and that in the statement 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
… Naming of the product concepts in
```

5.4 the body by macro the product concepts in the body

the macro one is several kinds: concepts named and destroyed

- automatically at the end of the execution of the macro-command. One thus should not any more need some thereafter and it should in particular be taken care that the product concept 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 begins with __(double underscore) Example: __a=CALC_MATR_ELEM (=MODELE

MODELS

```
) 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 exists any more, neither within the space of names of the supervisor, nor in the jeveux base. concepts named automatically

- and preserved in the jeveux base at the end of the macro-command . To indicate that it is about a concept of this kind, it is enough to give him a name which begins with _(simple underscore) Example: _a=CALC_MATR_ELEM (=MODELE

MODELS)

```
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 does not exist within the space of names of the supervisor, it is on the other hand present under this name in the jeveux base. This kind of object answers the typical locations

where the product concept by the macro one “depends” on a concept upstream which will have to be always present: for example a model compared to a mesh , `one` matr_asse compared to numerical `a _ddl.` concepts intended `to become`

- product concepts of macro. To indicate that it is about a concept of this kind, it is necessary to call the DeclareOut method of the object macro coil `with, like` arguments, the name `of` the variable back from the command and the object resulting from the key words of the macro-command. Example: one associates the local variable to the key word mm MATRICE the macro one : self.DeclareOut ("mm ", MATRICE) mm

```
=ASSE_MATRICE (...........)
the concept of output the macro one
```

- (necessarily single) is treated in a similar way by indicating the concept of output self.sd. self.DeclareOut ("mm", self.sd) mm=ASSE

```
_MATRICE (...........) mm will become
the concept of output
```

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

5.5 macro operation of the particular and

different supervisor being according to whether a command is carried out in the command set or a macro-command, the processing 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
: . /code/
In /code/, one has access to
resu. Why
```

? because as of the call to the command, 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 of this way, it is more comforme 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 product concept in this macro. One will make then: try: . __resu = STAT_NON_LINE
(...) except

```
NonConvergenceError
: . 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 a assd object which gives access

data structure FORTRAN of name .9000027. In this case, resu. Liste_Vari_Acces () will function. If the command emits an error "F",

a assd object nevertheless will be recovered but data structure 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). To reach the product concepts before macro

5.6 D" general way, all the concepts

necessary to operation the macro one must forward by its key words. 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 data structure as starter. To obtain the concept from a name, it is necessary to use the method get_concept of the MACRO object. Maybe in `the body` the macro one: object `= self.get_concept` (name) In the same order

```
of idea, one can need
```

all the objects available (for example, STANLEY proposes the choice among the already calculated `concepts`). For that, one recovers the context like this: dictionary = self.get_contexte_courant () the method

```
get_concept gives access to the concepts
```

recorded `near` the command set. A temporary concept created in a macro-command but which was hidden with the user (named for example _9000028), cannot be recovered directly. `It is necessary to create` a reference towards data structure FORTRAN by utuilisant get_concept_by_type. Example: object = self.get_concept `_by_type ("_9000028`", mesh

```
_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. To number the macro one In the displays of the file of

5.7 messages, all

the commands 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 command "girl" of the macro command because this method also initializes the total measurement of TEMPS CPU for the macro one. To treat the errors As for a command in FORTRAN,

5.8 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 printing of messages). With this intention, it is necessary to import this method since the modulus of the utilities Python. Example: from Utilitai.Utmess importation UTMESS… UTMESS

("F", "LOADS

```
_4", valk= " DX", vali= (2, 88))
The first
argument indicates the nature of the error or alarm
```

, the second and the specifies the identifier of the message in the catalog of messages following arguments (vali, valr, valk) make it possible to provide integer, `real variables` or character strings to supplement the message. The displays As much as that is possible, the messages

5.9 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 catalog of messages , it is possible to directly print text on the message file or RESULTAT by employing the method displays `modulus` aster `. The use` of the command `print is strongly disadvised`

. Example of use `of` aster.affiche: importation aster…

aster.affiche (“MESSAGE”, _de_caracteres `chains`

```
) the first
argument
is worth "MESSAGE" or "RESULTAT"
```

according to the target file. To identify the product concepts by the macro one In certain circumstances

5.10 , it is necessary to determine if a concept

is produced by macro itself or were produced by a preceding command. This is possible while testing if the concept in question is present or not in the list of the product concepts 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
```
macro # it is necessary to call
   command NUME_DDL lnume = 1 else: # the concept
   nume_ddl already exists. lnume=0 Attention
   , sdprods

   does not contain the product concept turned over
```

by the macro one `which` is in the attribute sd of coil. Dynamic creation of commands: many variable key words

5.11 , contextual contents In certain cases, according to the value of the options, the same command

will be called with various key words or of the different arguments. To treat this situation and to generate the command dynamically, one builds a dictionary containing the keywords to be written which is then transmitted in argument of the command, preceded by the characters “**”. The dictionary “is then unfolded”, the words are the arguments (keywords), followed by the contents of the word, behind the sign “=”. This dictionary Python can be built as

the examination of the options. Example: moscles= {} moscles [“INFO”] = 2 motscles [“CREA_GROUP_NO

”]

```
= [] for grma
in GROUP_MA_BORD: motscles

["CREA_GROUP_NO"]. (
_F (GROUP_MA = grma)) _nomlma suspends
    = DEFI_GROUP (reuse = _nomlma MAILLAGE = _nomlma, ** motscles

) the dictionary  motscles contains
                                              the definition of
                                              INFO and
```

a list of key words `factors` CREA_GROUP_NO. For `the example` , the list is a simple recopy of `GROUP_MA_BORD` built by addition successive of an element. Call `to an external` code If one wishes to carry out in macro

5.12 third code by the command

EXEC_LOGICIEL or by means of the Utilitai.System modulus, one must `recover` the standard path of the appealable `software` (called directory of the tools). Since python by the method repout of the modulus aster. importation os.path importation aster path = aster.repout `()` `miss` 3D = os.path.join (path , "

```
miss3d") EXEC_LOGICIEL
(…
, LOGICIEL = miss3d,
…) Rules of programming the compliance
with rules of programming is  necessary
```

5.13 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 by line (not of instruction on the same one 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 character string Doc. of the functions is compulsory to describe what they do. In small letters
- to name the variables, the classes with a capital letter with each key…
- 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). Key words 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 key words. Except the product 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 `makes` a diagnosis of the respect of conventions… Consideration on the use of `the macros` in Python Definition of macro except catalog

6 the standard method to add the definition

6.1 of macro in Python for an execution

of Code_Aster is to add it in the catalog 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 catalog.

With this intention, it is enough to create a modulus Python containing the definition of macro by adding at the top of the modulus the importation of the variables of the catalog. Simplified example: from Cata.cata importation * def ma_macro_prod (coil,…): ..... def ma_macro

_ops (coil,…): ….

```
MA_MACRO=MACRO (nom=' MY
_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 modulus 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 `(...)`