# Process control supervisor and language

**Summary:**

This document describes the role and operation:

- of the supervisor who ensures the piloting of an execution of *Code_Aster* ;
- and of the process control language which ensures the communication between the user and the code.

# Contents

# 1 Introduction

The role of the supervisor is to ensure the command of the course of operations in the course of execution of a program. The instructions of execution are generally provided by the user. This requires a formalization of the communications between the code and its owner, it is it *process control language*.

The language Python is employed to write the catalogue of orders, the supervisor and the command files user. For the command files, that makes it possible to discharge the supervisor from the task of syntactic analysis, reserved for Python itself.

A command file is a succession of call to functions Python (orders), defined in the catalogue of orders. These functions have arguments of entry: keywords and their contents, and of the arguments of exit: produced concepts. The user who composes his command file must thus subject himself to the general syntax of Python (parenthesizing, indentation…) and with the rules imposed by the catalogue of orders (the provided arguments are coherent with until the function waits).

For a first making of contact with the code, the reader will be able not to approach chapter 2.

# 2 Mechanism general of operation of the supervisor

## 2.1 General architecture

Basic elements concerned during an execution of a calculation *Aster* are:

- the command file, provided by the user,
- the catalogue of orders: it is a module python of name `catastrophes` placed in the package `Catastrophes`,
- the SUPERVISORY high level object,
- object JDC created by this last and which is finally carried out.

The SUPERVISORY object is a python object which analyzes the options transmitted on the command line, imports the catalogue of orders, created object JDC starting from the command file and carries out this one.

Object JDC (name for Command set) is a python object created by the SUPERVISORY object starting from the text of the command file and of the module catalogues orders. It contains the STAGE objects. Object JDC is representative of the command file user.

The STAGE objects are representative of each call to orders Aster in the command file. Each STAGE object is called after the order that it reference, the list of the active keywords and their values, the type and the name of the produced concept.

Construction then the execution of object JDC start the following actions:

- syntactic analysis of the command file user: it is on this level that syntax python is checked (brackets, commas between keywords, indentation…). The detection of an error (`SyntaxError Python`) cause the stop of the execution of Aster. The first error is fatal: one does not search the following errors,
- construction of the stages: that consists in creating a STAGE object for each call to an order Aster in the command file. This object is recorded at JDC which manages the list of the stages and the related concepts,
- checking of each STAGE: if the call to an order in the file user is incoherent with the catalogue of orders, a report is displayed and the execution is stopped on this level. It is the semantic checking,
- execution itself of the orders: for each stage taken in the order, call to high level routine FORTRAN (op0nnn.f) corresponding.

## 2.2 Total execution or step by step

A command set can be built and carried out according to two modes:

- the total mode for which **all** the stages of the command set are initially built then carried out in their order of appearance. This mode is chosen by the keyword PAR_LOT=' OUI' in the ordering of starting BEGINNING,
- the mode step by step for which each stage is immediately carried out after its construction. This mode is chosen by the keyword PAR_LOT=' NON' in the order BEGINNING.

If the user does not specify anything in the order starting, total mode (PAR_LOT=' OUI') is retained. These two modes present each one their advantages and disadvantages.

The total procedure guarantees to the user that all its file is semantically correct before starting calculations which could fail or not converging. It would be indeed a shame to stop in fatal error after a long resolution because of a keyword forgotten in an order of postprocessing.
That also means that all the stages of the command set are built and stored. If several thousands of stages are reached, that can become very consuming memory and this mode is not then advised any more.

The mode step by step builds a stage only after having carried out the preceding one. It thus detects only the semantic errors of the pending order and presents the disadvantage described above. It however makes it possible to exploit a result calculated (in a concept) in the command file for, for example, to place conditional instructions there.
In this mode, the stage carried out is released at once from the memory. The memory used is then independent amongst stage to carry out.

Here an example of loop with a criterion of stop on the value of a calculated size, stored in the concept of the type table : RELV [K]. If for example an obligatory keyword misses in the call to POST_RELEVE_T, that will be detected only after the complete execution of the first MECA_STATIQUE. On the other hand, the mode step by step makes here possible the assignment of the variable SYY since the concept RELV [K] was completely calculated at the time when the supervisor carries out this line.

```
BEGINNING (PAR_LOT=' NON')

RESU= [Nun] *10
RELV= [Nun] *10

for K in arranges (1.10):

   RESU [K] =MECA_STATIQUE (…)
   RELV [K] =POST_RELEVE_T (…)
   SYY=RELV [K] ['VMIS', 4]

   yew SYY < criterion:
     station-wagon

END ()
```

It should be noted that the choice of a procedure conditions the order in which the analysis will proceed **semantics** (STAGE by STAGE or overall for all the JDC). But, in both cases, the analysis **syntactic** python is always made as a preliminary for all the command file.

**Note:**

*EFICAS can exclusively generate and read again only command sets containing orders ASTER, without other instructions python; this independently of the mode PAR_LOT chosen.*

## 2.3 The construction of the stages

During the construction of each STAGE object, one checks his semantic coherence with the catalogue of the order to which it refers. Any detected error is consigned in a report which, in total procedure, is delivered after the analysis of all the command file.

Semantic examples of checks:

- respect amongst arguments of the keywords,
- respect of the type of argument,
- membership of an argument to a list of possible values,
- exactitude of the orthography of a keyword or a keyword factor,
- compliance with the rules of exclusion or implication between keywords,
- presence of the obligatory keywords.

This stage, if the order is an operator and produces a concept, this one is typified. The supervisor checks that of a the same concept name was not already defined, or if it is employed again, that the order authorizes it.

## 2.4 Treatment of the macro-orders

An macro-order, considering the user, is an ordinary order. In fact, it does not call a high level routine FORTRAN directly but generates other orders.

Two types of macro-orders exist:

- macros in Python,
- macros supervisory: they are the special orders (BEGINNING, FORMULA, INCLUDE, INCLUDE_MATERIAU, CONTINUATION) who require a treatment the level of their construction.

As well as the JDC itself, the call to an macro-order produces an object father (of type MACRO-ETAPE) which contains objects son: the stages which the macro one generates, even others macros.

An macro-order of the JDC is first of all treated like the other orders (syntactic checking, construction of the macro stage). Then it “is built” by application of the method python Build on object JDC. After its construction, the stages of the orders produced by the macro one are substituted at the stage of macro itself, for later execution.

It is important to note that the phase of construction of the macro-orders proceeds right before their execution, and not at the time the total master key on the command file in mode PAR_LOT=' OUI'. That has two consequences:

- EFICAS analyzes the syntax of the macro-order itself, but not that of its under - orders.
- One can on the other hand exploit, in the programming of the macros, the data previously calculated and repatriated within the space of names python, without having to impose the mode PAR_LOT=' NON' with the user the macro one.

## 2.5 Procedures of starting

The procedures of starting available are:

BEGINNING (cf [U4.11.01] and CONTINUATION [U4.11.03])

At least one of these two procedures must be obligatorily present in the command file. No other order Aster must precede them. If it is the case or if none is present, the execution will be stopped as of the creation of the JDC. These are the procedures which contain information on the procedure (PAR_LOT=' OUI' or 'NOT') who conditions the procedure of the orders which follow.

In fact macro-orders supervisor, with their construction, call routines FORTRAN making it possible to initialize calculation by the following tasks:

- “connection” of the logical units of the standard files,
- opening of the databases,
- reading of the catalogue of elements.

The first task consists in putting in correspondence numbers of logical units of standard files of input/output (message, error, result).

The second task consists to define and open the databases (file of direct access used by the manager of memory) in accordance with the instructions of the user, who can redefine parameters of these files (see documents [U4.11.01] and [U4.11.03] on the procedures of starting). One calls for that the routines of initialization JEVEUX (see document [D6.02.01] the Manager of memory, JEVEUX).

The sequence of the orders to be carried out ends obligatorily in the order END. The text which follows END must be commentarized (i.e. begin with #). For a file included, it is the order RETURN who marks the end of the instructions that ASTER must take into account.

**Note:**

> *In interactive mode, seizure of the manual controls, not to put of order END and to pass the argument - interact on the command line of tender of the job.*

## 2.6 Links with EFICAS

The core of the supervisor is common with Eficas, the editor of command files Aster. At the time of the edition of a command file, this one carries out the syntactic analysis and the checks of coherence of the concepts by construction of the JDC and of its STAGE objects. Eficas does not carry out of course the task of construction of the macro-orders which would require the source code of*Aster*.

# 3 The process control language

## 3.1 Python and the language process control

A command file for *Code_Aster* is exclusively made up of instructions Python. The first of the constraints is thus to conform to the rules of this language. One will be able to read the tutorial Python (www.python.org) or many books of introduction to Python for more detail, but it is not necessary for the use of Aster.

A command file can contain instructions python of two natures: orders *Aster* and… any other instruction python. Indeed, a command file is a program python except for whole and one can place in particular there structures of control (loops), tests (yew), digital calculations, calls to functions the pre one and postprocessing.

Within the framework of a “classical” use of the code where the command file contains orders Aster exclusively, the two rules specific to Python to be retained are:

- Pas d' indentation on the first line of declaration of an instruction.
  ```
  e-mail = LIRE_MAILLAGE ()
  ```
  One should place neither white, nor tabulation before the character string e-mail.

- Arguments of the functions, in other words the keywords of the orders, are separated by commas; they are composed of a keyword, sign “=”, contents of the keyword.

**Important:**

Editor EFICAS allows to produce only command files of this type: containing exclusively orders ASTER, without another instruction Python. To use EFICAS guarantees primarily three things:

- the produced file will have a correct syntax python,
- the produced orders will be coherent with the catalogue of orders,
- the produced concepts will be correctly connected (not of use of a concept without it being created by a preceding order).

The user having composed his command file will thus be safe from a stop with the execution with the reason for a problem of syntax.

## 3.2 Concept of concept

Definition: one calls *concept* structures of data Aster, that the user can handle and name. These concepts are typified at the time of their creation and could be used only as argument of entry of the type corresponding in a further order.

The concept of concept thus makes it possible to the user to handle objects symbolically and independently of their internal representation (which it can not know). Moreover, the python object indicated by the name of the concept does not contain any other information but its type, its class with the direction python (cf Doc. D). Its name, transmitted by the supervisor to FORTRAN, makes it possible Aster to find the corresponding structure of data in the total base. But it is not possible to have visibility of the structure of data since the command file. For example, the following instructions do not make it possible to print the structure of data of the type `grid` and of name `e-mail` :

```
mail=LIRE_MAILLAGE ()
print e-mail
```

but generate the following message:

```
<Cata.cata.maillage_sdaster object At 0x593cad0>
```

There is an exception to this rule: tables. Indeed, an artifice of programming makes it possible to simply recover contained information in structure of data `TABLE` by handling this one like a table at two entries:

to print a value: `print resu ['DX', 1]`
to assign it to a variable: `value = resu ['DX', 1]`

That supposes of course that the structure of data `resu`, of type `TABLE`, was already calculated at the time when this instruction is met: thus in procedure step by step (`PAR_LOT=' NON'`).

**Notice lexical:**

The names of concepts should not exceed 8 characters. The alphanumerics are licit (small letters and capital and figures not placed in first position) as well as the underscore '_'. Breakage is important: the concepts "E-MAIL" and "E-mail" could be used in the same command file and will be regarded as different… it however is disadvised for the legibility of the file!

## 3.3 Possible operations

The structure of the process control language is appeared as a linear succession of instructions. In addition to the instructions python other than of the orders Aster, of which it is not question for the moment, three natures of instructions (or orders) are available:

- the operator who carries out an action and who provides one `concept product` of a preset type exploitable by the following instructions in the command set,
- the procedure which carries out an action but does not provide a concept,
- the macro-order which generates a sequence of instructions of the two preceding types and which can produce zero, one or more concepts.

Typically, an operator will be an ordering of assignment or of resolution, a procedure will be an ordering of impression (in a file).

From the syntactic point of view an operator presents himself in the form:

```
nomconcept = operator (arguments…)
```

Whereas a procedure arises in the form :

```
procedure (arguments…)
```

The syntax of an operator or a procedure is described in the following paragraph.

## 3.4 Rules on the concept produced by an operator

### 3.4.1 Basic principle

With each execution of an operator, this one provides a new produced concept of the preset type in the catalogue of order.

*The concepts appearing in argument of entry of the orders, are not modified.*

### 3.4.2 Concept produces and re-used concept

One calls re-used concept, a concept which being produced by an operator, is modified by a new occurrence of this operator or by another operator.
The use of a re-used concept is not possible, like exemption of the Basic principle that in two conditions:

- authorization given, by the catalogue and the programming of the order, to use reusable concepts for the operator: the attribute `reentrant` catalogue is worth `'O'` or `'F'`,
- request clarifies of the user of the re-use of a concept produced by the attribute `reuse=nom_du_concept` in the arguments of the orders which allow it.

### 3.4.3 Checks carried out by the supervisor on the concepts produced

- Produced concept respecting the basic principle:
  The supervisor checks that the name of the produced concept is not already allotted by one of the preceding orders, in particular by an ordering of an execution preceding in the case of one `CONTINUATION` or of one `INCLUDE`.
- Concept used in re-use:
  The supervisor checks that:
  - the name of the produced concept is already well allotted.
  - the operator is well entitled to accept re-used concepts,
  - the type of the concept is in conformity with the type of `concept product` by the operator.

Examples with accompanying notes:

```
BEGINNING ()
concept=operator ()                 #   (1)     is correct: one definite the concept,
concept=operator ()                 #   (2)     is incorrect: one tries to redefine it
                                    #           concept but without saying it,
concept=operator (reuse = concept)  #   (3)     is correct, if the operator accepts
                                    #           existing concepts and if the type is
                                    #           coherent; it is incorrect if the operator
                                    #           does not accept them.
END ()
```

In fact a concept can be created only once: what means to appear sign on the left = (equal) without reuse that is to say employed in the arguments of the order.

In the case of a re-use, to again specify the name of the concept behind the attribute reuse is redundant; more especially as the supervisor checks that the two names of concept are identical.

**Note:**

*One can destroy a concept, and thus re-use his name then.*

## 3.5 Body of an order

### 3.5.1 Introduction

The body of an order contains the “variable” part of the order. The declarations are separated by commas and except for the attribute reuse mentioned above, they all are of the form:

[mot_clé] **=** [argument]

The keyword is necessarily a keyword of the pending order, declared in the catalogue of this one.

### 3.5.2 Keyword

A keyword is a formal identifier, it is the name of the attribute receiving the argument.

Example: MATRIX **=**…

**Syntactic remarks:**

- *the order of appearance of the keywords is free, it is not imposed by the order of declaration in the catalogues,*
- *the keywords cannot exceed 16 characters (but only the first 10 characters are meaning).*

There exist two types of keywords: simple keywords and the keywords factors which differ by nature from their arguments.

### 3.5.3 Argument of a simple keyword

#### 3.5.3.1 The type of the arguments

The basic types recognized by the supervisor are:
- entireties,
- realities,
- complexes,
- texts,
- logics,
- concepts,
- as well as the lists of these types of bases.

The entireties and realities correspond exactly to the equivalent types in python.
- Optional simple keyword expecting a reality:
  Catalogue : VALE = SIMP (statut=' f', typ = 'R'),
  Command file : VALE = 10. ,
- Optional simple keyword expecting an entirety:
  Catalogue : INFORMATION = SIMP (statut=' f', typ = 'I'),
  Command file : INFORMATION = 1,

The representation of the complex type is a “tuple” python containing a character string indicating the mode of representation of the complex number (left real and imaginary or module-phase) then the digital values.

```
Catalogue                 :  VALE_C = SIMP (statut=' f', typ = 'C'),
Command file      :  VALE_C = ('IH', 0,732, -0,732),
Command file      :  VALE_C = ('MP', 1. , -45. )  ,
```

The two notations are strictly equivalent. In notation `'MP'`, the phase is in degrees.

The standard text is declared between simple dimensions. Breakage is respected. However, when a keyword must take a value in a preset list in the catalogue, the use wants that this value is today always in capitals.

```
Catalogue                 :  ALL=SIMP (typ= 'TXM', into= ('YES', 'NOT')) ,
Command file    :  ALL= 'YES',
```

Breakage is important and, in the context above, the following command line will fail:

```
Command file    :  ALL= 'yes',
```

The logical type is not used today in the catalogue of orders.

The concept is declared simply by its name, without dimensions nor quotation marks.

#### 3.5.3.2 Concept of list

**Caution** :

The word “list” is an abuse language here. It is not the type “lists” python but rather of tuples, within the meaning of python: different the items is declared between an opening bracket and a closing bracket; they are separated by commas.

The lists are homogeneous lists, i.e. whose elements are of the same basic type. Any basic type can be used in list.

Examples of list:

```
list of entireties      (1, 2,3,4),
list of text            ('this', 'is', 'one', 'list', 'of', 'text'),
list of concepts        (resu1, resu2, resu3),
```

User friendliness:

It is allowed that a list reduced to an element can be described without bracket.

Example of erroneous list:

Heterogeneous list of entirety and reality

```
(1, 3.4.)
```

### 3.5.4 Keyword factor

Certain information cannot be given overall (in once in the order), it is thus important to envisage the repetition of certain keywords, to be able to affect different arguments to them. The keyword factor gives this opportunity; under a keyword factor, one will thus find a set of keywords (simple), which could be used with each occurrence of the keyword factor. That makes it possible moreover to improve the legibility of the command file by gathering keywords which share a common direction: for example various parameters of the same material.

Contrary to the simple keyword, the keyword factor can receive one type of object: the supervisory object " _F ", or a list of this one.

That is to say the keyword factor has only one occurrence and one can write for example, with the choice:

```
IMPRESSION    =      _F (   RESULT =  resu,  UNIT = 6),
or
IMPRESSION    =   (   _F (   RESULT =  resu,  UNIT = 6),),
```

In the first case, the keyword factor IMPRESSION receives an object _F, in the other, it receives a singleton. Attention with the comma; in python, a tuple with an element is written: (element,)

That is to say the keyword factor has several occurrences, two in this example:

```
IMPRESSION    =   (   _F (   RESULT =  resu1, UNIT = 6),
                      _F (   RESULT =  resu2, UNIT = 7) ),
```

The number of occurrence (minimum and/or maximum) expected of a keyword factor is defined in the catalogue of orders.

**Concept of value by default**

It is possible to make affect by the supervisor of the values by default. These values are defined in the catalogue of orders and not in FORTRAN.

There is no distinction from the point of view of the routine associated with the order between a value provided by the user and a value by default introduced by the supervisor. This appears during the impression of the orders user by the supervisor in the file of messages: all the values by default appear in the text of order, if they were not provided by the user

**Recall** : one cannot give value by default to a concept.

# 4 Definition of values and evaluations of expressions

It is possible to assign values to variables python in order to use those like arguments of simple keywords: these variables are called parameters in EFICAS. They can contain values whole, real, complex, texts or lists of these types.

Example:

```
Young =  2.E+11
chechmate    =  DEFI_MATERIAU(ELAS = _F (  E = Young, NAKED = 0.3))
```

At the end of the execution, the context python is saved with the base. Thus, in the continuation which will follow, the parameters will be always present, with their preset values, just like the concepts ASTER.

It is possible to carry out operations in python on the simple arguments of keywords:

```
Pisur2 =  pi/2.
chechmate    =  MA_COMMANDE   (VALE = Pisur2)
```

or:

```
VAr    =  'world'
chechmate    =  MA_COMMANDE   (  VALE   = pi/2. ,
                          VALE2  = Pisur2+cos (30.),
                          TEXT   = 'hello' +var      )
```

# 5 Use of python in the command files

It is not necessary to know the language python to use *Code_Aster*. Indeed, with the help of some basic rules to respect on the indentation and parenthesizing, only the knowledge of the process control language describes in the catalogues of order is necessary. And still, EFICAS makes it possible to be exempted to resort to the catalogue or the paragraph “syntax” of the orders by graphically proposing the keywords to be informed.

However, the advanced user will be able to use cheaply the power of the language PYTHON in his command file, since this one is already written in this language.

The four principal uses can be: the writing of personalized macro-orders, the use of general instructions python, the importation of useful modules python, the recovery of information of the structures of data *Code_Aster* in variables PYTHON.

**Note:**

> *If one wants to use French characters accentuated in the command file or the modules imported, it is necessary to place the following instruction in first or second-row forward of the file:*
>
> `# -*- coding: Iso-8859-1 -*-`
>
> *In python 2.3, the absence of this line causes a warning which will become an error in python 2.4; in ASTER, it is systematically an error.*

## 5.1 personalized Macro-orders

See the document [D5.01.02]: “To introduce a new macro-order”.
The personalized macro-orders are very easy to program. They can be used for capitalizing recurring diagrams of calculation and thus constituting a tool-trade. It is strongly advised to take as a model the existing macro-orders: package `Macro` in the repertoire `bibpyt`.

## 5.2 General instructions PYTHON and useful modules

The advanced users can gain great profit from the use of loops (for), of tests (yew), the exceptions (try, except) and in a general way of all the power of the language PYTHON directly in their command file. The list of the uses is impossible to establish exhaustively. Many examples are present in the cases tests of the base of tests. One can for example make adaptation of grid while placing the sequence calculation/mending of meshes in a loop, to establish a criterion of stop of the iterations by a test on a computed value.

To consult the following paragraph dedicated to the “particularized” exceptions Aster.

In a loop, if an already existing concept is recreated, it is necessary to think of destroying it as a preliminary by the order `TO DESTROY`.

Other various features of python interesting for the user of *Code_Aster* can be:

- the read-write on file,
- digital calculation (for example by using Numerical Python),
- the call via the module `bone` with the language of script, and in particular the launching of a third code (`os.system`)
- the handling of character strings
- the call to graphic modules (grace, gnuplot)

## 5.3 Exceptions PYTHON of the module aster

The mechanism of the exceptions Python is very interesting, it authorizes for example "to try" an order then to take again the hand if this one "plants" while raising a particular exception:

```
try:
    block of instructions
except ErreurIdentifiée, message:
    block of instructions carried out if ErreurIdentifiée occurs…
```

In the command file, one can use this mechanism with any exception of the modules Python standards.
One also has exceptions suitable for *Code_Aster* (with the module `aster`), divided into two categories, the exceptions associated with the errors `<S>`, and that associated with the errors `<F>`.

### 5.3.1 Interception of the <S> errors

In the event of error `<S>`, the error is identified by one of these exceptions:

| | |
|---|---|
| `aster.NonConvergenceError` | in the event of nonconvergence of calculation, |
| `aster.EchecComportementError` | problem during the integration of the law of behavior, |
| `aster.BandeFrequenceVideError` | pas de mode found in the waveband, |
| `aster.MatriceSinguliereError` | singular matrix, |
| `aster.TraitementContactError` | problem during the treatment of the contact, |
| `aster.MatriceContactSinguliereError` | singular matrix of contact, |
| `aster.ArretCPUError` | stop by lack of time CPU. |

All these exceptions derive from the exception `<S>` general which is `aster.error`. What means that `except aster.error` intercept all the exceptions quoted below. It is nevertheless always preferable to specify with what a error one expects!

Example of use:

```
try:
    resu = STAT_NON_LINE (...)
except aster.NonConvergenceError, message:
    print 'Stop for this reason: %s' % str (message)
    # One knows that one needs much more iterations to converge
    print "One continues by increasing the iteration count."
    resu = STAT_NON_LINE (reuse = resu,
                          …,
                          CONVERGENCE=_F (ITER_GLOB_MAXI=400,),)
except aster.error, message:
    print "Another error occurred: %s" % str (message)
    print "Stop"
    from Utilitai.Utmess importation UTMESS
    UTMESS ('F', 'Example', 'unexpected <S> Error')
```

### 5.3.2 Interception of the <F>atales errors

In the event of fatal error, the behavior is modifiable by the user.
By default, the code stops in error `<F>`, one can see the increase of error (call of routines FORTRAN) in the output file.
If it is wished, the code can raise the exception `aster.FatalError`, and in this case (as for an error `<S>`), if the exception is intercepted by one `except`, the user takes again the hand, if not the code stops (not increase of error FORTRAN).
This choice is given in the orders `BEGINNING`/`CONTINUATION`, keyword factor `ERROR` (cf [U4.11.01] and [U4.11.03]) and constantly by the method `aster.onFatalError`.

The latter called without argument turns over the current behavior in the event of fatal error:

```
comport = aster.onFatalError ()
print "Behavior running in the event of fatal error: %s" % comport
```

and allows to define the behavior which one wishes:

```
aster.onFatalError ('EXCEPTION')    # one raises the exception FatalError
```

or

```
aster.onFatalError ('ABORT')        # one stops with increase of error.
```

### 5.3.3 Validity of the concepts in the event of lifting of exception

At the time an exception is raised and intercepted (by except), the concept produced by the order in which the error occurred is returned such as it is, the order not having finished normally.
In certain cases, in particular after a fatal error, it may be that the produced concept is not usable; to use then TO DESTROY to remove it.
In the same way, if one wishes to re-use the name of the concept to create new, it is necessary TO DESTROY that obtained in try.

In the event of error <S> liftings in STAT/DYNA_NON_LINE, the concept is generally valid, and can be re-used (keyword reuse) to continue calculation with another strategy (as in the example quoted previously).

Lastly, before returning the hand to the user, the objects of the volatile base are removed by a call to JEDETV, and Jeveux marks it is repositioned to 1 (with JEDEMA) to release the objects brought back in memory.

### 5.3.4 Precautions for use of the exceptions

Two exceptions aster.error and aster.FatalError are independent (none derives from the other), which means that if one wishes to take again the hand in the event of <S> error and of <F> error:

- it is necessary to activate the lifting of exception in the event of <F> error with aster.FatalError ('EXCEPTION') or in BEGINNING/CONTINUATION.
- the two exceptions should be intercepted:
  ```
  except (aster.FatalError, aster.error), message:  …
  ```

It is disadvised using " except: " without specifying with which exception one expects (it is a general rule in Python independently of the exceptions Aster). Indeed, treatment carried out under except little chance has to be valid in all the cases of error.

In the same way, as in the example set higher, it is preferable to use the particularized exceptions NonConvergenceError, ArretCPUError,... that exception moreover high level aster.error ; always in the idea of knowing exactly what occurred.

## 5.4 Recovery of computed values in variables PYTHON

To exploit the language PYTHON in its command file is not interesting that if one can conditionally launch actions according to what the code calculated.

Certain footbridges exist between python and the structures of data calculated by the code and present in memory JEVEUX. Others remain to be programmed; this is a field in evolution and future developments are expected.

It is essential to understand that to recover calculated data requires that the instructions involving their obtaining were indeed carried out as a preliminary. In other words, it is essential to carry out the code in mode PAR_LOT=' NON' (keyword of the order BEGINNING). Indeed, in this case, there is no total

analysis of the command file, but each instruction is carried out sequentially. When one arrives on an instruction, all the concepts it preceding were thus already calculated.

Here some access methods to the structures of data. The list is nonexhaustive, to refer to documentation [U1.03.02].

| Structure of data | Method | Standard turned over python | Turned over information |
|---|---|---|---|
| listr8 | LIST_VALEURS | list | List of the values |
| grid | LIST_GROUP_NO | list | List of the groups of nodes |
| | LIST_GROUP_MA | list | List of the groups of meshs |
| table | […] | reality | Contents of the table |
| function | LISTE_VALEURS | list | List of the values |
| result | LIST_CHAMPS | list | List of the computed fields |
| | LIST_NOM_CMP | list | List of the components |
| | LIST_VARI_ACCES | list | List of the variables of access |
| | LIST_PARA | list | List of the parameters |
| cham_no | EXTR_COMP | post_comp_cham_no | Contents of the field in a table |
| cham_elem | EXTR_COMP | post_comp_cham_el | Contents of the field in a table |
| Any object JEVEUX | getvectjev | list | List of the objects of the vector jeveux |

## 5.5 Dynamic example of construction of keywords factors

In the case of a keyword very repetitive factor to write, the user can want to compose his contents by script, in a list or a dictionary, which it will then provide to the keyword factor. The example below watch three ways of writing the same keyword factor.

```
BEGINNING (PAR_LOT=' NON')

ooo= [ _F (JUSQU_A=1., PAS=0.1),
       _F (JUSQU_A=2., PAS=0.1),
       _F (JUSQU_A=3., PAS=0.1)]

ppp= [_F (JUSQU_A=float (I), PAS=0.1) for I in arranges (1.4)]

qqq= [{'JUSQU_A': float (I), 'NOT': 0.1} for I in arranges (1.4)]

rrr= [_F (** arguments) for arguments in qqq]

li1=DEFI_LISTE_REEL (  DEBUT=0.,
                     INTERVALLE=ooo
                     / or
                     INTERVALLE=ppp
                     / or
                     INTERVALLE=rrr
                     )

END ()
```