Maintenance of the supervisor of *Summarized*

Code_Aster:

This document is with the use of the developers of the supervisory core of Aster. It clarifies the interpretation of the data file of the user (construction of the command set) and the sequence of the executions.

Nota bene important:

This document was produced for version 6 of Code_Aster and was not updated since. It is preserved here for archive.

Contents

1 numerical

Introduction Study

to carry out a numerical study with *Code_Aster*, the end-user starts the features of the code and provides the necessary information to the execution of these features.

A functionality is selected via a command. A command is initially a name i.e. a character string or an external **identifier** (known of the user) of the functionality. It also incorporates attributes and initially the internal **identifier** of the functionality (exploitable by code FORTRAN).

The necessary information with launching and the execution of the command otherwise-known as the parameters of the digital processing, are data introduced by means of key words the identifier. For a given command, a certain number of parameters must be defined.

The description of the commands and the key words is carried out by the developers of *Code_Aster*. in a file called “catalogues commands”.

The end-user uses the code via a file called “command file”. He provides to it commands whose composition must be compatible with their description in the catalog of the commands. The commands are numerous but the number of associated data is much more important with possible combinations themselves very many.

Finally the computer code stores information (commands and key words) in an internal data structure called “command set”.

The role of the supervisor

the supervisor is the part of *Code_Aster* which manages the command set. In particular:

1) it imports the catalog of the commands in the memory Python,
2) it charges the commands and their key words in the memory Python,
3) it carries out the processing orders by command,
4) he provides - at the request of FORTRAN – the value of the parameters to the features of *Code_Aster*. stored in the memory Python. For that the supervisor proposes API C.

graphic interface Eficas

It is possible to define “manually” his command file, for example by means of a text editor. However, the user nonfamiliar with syntax of the commands which it handles but also the language python will be able to use Eficas.

The Eficas graphic interface is intended to the end-user of *Code_Aster*. It enables him to build valid commands with statically such validated associated key words, then to generate a command file bound for the code. The user has the assurance which the file produced by Eficas has a correct syntax.

The core

the sources common to the Eficas interface-graph and the supervisor of *Code_Aster* are organized (identified and gathered) in a specific space of development called the “Core”. The comprehension of these sources is obviously a requirement to undertake the maintenance of the two tools which use them.

The main actors

the principal types of actors evolving in the environment of Eficas and the supervisor are listed Ci - below.

1) The end-user: he rather knows the physical part of the problem to be solved and he has as a task to introduce valid data into the code, of launching computations and stripping the results; he can use the Eficas graphic interface for that.
2) The developer of features in *Code_Aster* : it practice primarily programming FORTRAN of the numerical algorithms and, in the case of a macro-command, the drafting of script corresponding Python; it enriches and/or modifies the catalog of command and it C uses the API one of the supervisor to recover in his code FORTRAN, the values provided by the end-user.
3) The developer of the supervisor in *Code_Aster*: it writes scripts of reading and interpretation of the catalog of commands and the command set; it also deals with the modulus python `aster` writes in C (`astermodule.c) which` serves as API to the supervisor.
4) The person in charge of the maintenance of the code, the environment of use and management of configuration: he centralizes the sources. Besides code FORTRAN, these sources understand those of the catalog (Python) commands, scripts (Python) of the supervisor and those of the modulus aster .
5) The developer of the Eficas graphic interface develops a Man-Machine Interface. Its role is to conceive and program dialog boxes as well as sequences of event allowing the taking into account of the requests of the end-user and the checking – at every moment - validity of the command set in the course of construction. In

this document, one rather focusses on the principles of the design of the core that on the detail of scripts which one cannot save the examination. Covered

issues

the first chapter proposes a summary definition of the terms used abundantly in the continuation of the document.

Chapter 2 2 few conventions and the organization of the source files in directories: packages. It also describes the environment necessary for the development and the operation of the supervisor of Code_Aster *and* the Eficas graphic interface. In

chapter 3, some recalls are carried out on the language of script Python. They must direct the future person in charge of maintenance towards certain techniques that it will be necessary for him to control to carry out his task.

The important subject of the factory of command is evoked in chapter 4. 4 is a base necessary to the comprehension of all structure of the command set. Lastly,

an answer is given to chapter 5, 5some questions which one can provide that all future person in charge of maintenance will be able to be posed. Mode

1.1 definitions

“by batch”

the treatment of the commands user by the Code_Aster *can* be carried out according to two modes. In

the first mode, the file command is charged in memory to create the command set. This creation of the jdc makes it possible to validate syntax python (brackets, commas), syntax Aster (coherence with the catalog) and to validate the last concepts in argument. Finally after this checking, the command set is traversed to carry out the corresponding digital processings. This
first mode is called “mode by batch”. The end-user selects this mode by specifying the value “ YES” for key word PAR_LOT under the command debut . In

the second mode, the command file is charged in memory to create the command set. Then the stages (equivalent to the commands) are built and carried out sequentially.
The end-user selects this mode by specifying the value “ NON” for key word PAR_LOT under the command debut . By

default, the mode used is PAR _LOT=' OUI'. Operator

of Code_Aster

an operator is a unit of fascinating *Code_Aster* in load a functionality of the code. Concretely it is a subroutine FORTRAN whose name is numbered, for example the subroutine OP 0001 which charges a mesh in the memory with the application. The classification operators facilitates association between their internal representation (subroutine FORTRAN) and their external representation for the end-user (command). Order

a command is a character string identifying a numerical operator. It thus makes it possible to the end-user to start the execution of this operator from a data file called “command file”. There

exist 4 types of commands: OPER , PROC , MACRO and FORMULATE .

The developer of the numerical operator of the command defines - in the catalog of the command - the characteristics and those of the key word corresponding to the parameters of the numerical operator: its

1) name (the character string usable by the end-user), its
2) rules of composition in key words,
3) an explanatory French and/or English comment,
4) the word defining the handbook and the chapter devoted to this key word in the documentation of Code_Aster . *Order*

OPER Besides

the attributes enumerated above, an ordering of type OPER has the following attribute:

1) The type of the data structure Aster produced by the operator and turned over by the command; Order

PROC

an ordering of type PROC has the characteristic not to turn over a value. This characteristic put except for, it has the same attributes as an ordering of type OPER. MACRO

command

macro is a function written in Python – by the developer Aster - which calls commands – i.e. operators - of Code_Aster . *It* stores results which could be recovered via the supervisor.

The text the macro one can be public; in this case it is stored in a specific file of under - Macro directory. If it is deprived, it is placed or imported in the command file.

An ordering of MACRO type makes it possible to the end-user to use the macro one. For example the command ASSEMBLY `makes it possible` to launch the macro public assembly `_ops. Catalogue`

of a command

the catalog of a command is all the instructions Python describing the definition of the command i.e. the values assigned to the attributes of the command.

The catalog of a command is written by the developer of the numerical operator associated with the command. Catalogue

general

the general catalog is a file Python containing the description of all the commands otherwise-known as containing the catalogs of all the commands. Command set

the command set is the data structure - organized in a Python object – containing all of the furnished information by the end-user, in the command file. Command file

the command file makes it possible to the end-user to start the numerical operators carrying the features of Code_Aster *via* the commands. Data structure

Aster

a data structure Aster is an organization of data produced by a numerical operator of Code_Aster . It is identified by a type itself declared at the beginning of the catalog (cata.py`); what` makes it possible to use it – symbolically – in the command file although it is produced by FORTRAN. Simple

key word a simple

key word is a character string identifying a data used as starter by an operator (a numerical functionality of Code_Aster *).* A simple key word is thus defined inside a command of Code_Aster *.* The end-user

will be able to provide a value **to the parameter** of a command via the name of the simple key word corresponding in the command file. The developer

of functionality of Code_Aster *will define* as for him, the characteristics **of the simple** key word in the catalog of the command containing the simple key word: its name

1) (the character string usable by the end-user and by the operator numerical), the type
2) of the parameter (whole, real, text, concept,…), the statute
3) of the simple key word (optional or compulsory), the value
4) by default to be assigned to the parameter, the minimum
5) number of data which the end-user will have to provide behind the key word simple, the maximum
6) number of data which the end-user will have to provide behind the key word simple, an explanatory
7) French and/or English comment. The supervisor

of Code_Aster *charges* in the memory with the application, the characteristics of the simple key word, starting from the catalog of the commands. Then it of the command charges (and checks) possibly the value with the parameter starting from the command file provided to the application by the end-user. The numerical

operator of Code_Aster *questions* the supervisor via the API getvxx to recover the value of the parameter starting from the name of the key word. The supervisor turns over the value provided by the user or the value by default of the parameter. Factor key word

factor key word

is a semantically associated character string identifying a group of key word simple. Factor key word is defined inside a command. A command can factor key word contain several key words possibly optional factors., each containing itself of the simple key words of the same name. The end-user

will be able to define in his command file, factor key word by specifying his name then his value **i.e.** the value of the definite numerical parameters behind the simple key words of factor key word. The developer

of the functionality of Code_Aster *defines* the characteristics **of factor key word** in the catalog of the command containing factor key word: its name

1) (the character string usable by the end-user and by the operator numerical), its rules
2) of composition in simple key words, the statute
3) of factor key word (optional or compulsory), the minimum
4) number of repetition of factor key word, the maximum
5) number of repetition of factor key word, an explanatory
6) French and/or English comment, the word
7) defining the handbook and the chapter devoted to this key word in the documentation of Code_Aster *. Conditional*

block a conditional

block (a block), associates: simple

1) key words, conditional
2) key words factors, and
3) blocks. The occurrence

of the block in its command, depends on a condition expressed at the time of the definition of the command by the developer of the numerical functionality. The developer

of the operator corresponding to the command containing the block, specifies the characteristics of the block: its name

1) , its condition
2) , its rules
3) of composition in simple key words, an explanatory
4) French and/or English comment. The end-user

will be able by means of to give a value to the parameters of the processing the key words factors and the key words simple associates in the conditional block but without specifying the name of the block. Regulate

composition the composite

entities of the catalog of commands such as “command set”, orders, factor key word and conditional block, structure of other entities while possibly following one or more rules of composition among the following ones: AU_MOINS_UN

`rule`
AU_MOINS_UN `expresses` that one at least entities whose names passed in arguments must be present in the composite entity in which figure the rule. UN_PARMI

`rule`
UN_PARMI `expresses` that one and only one of the entities of the entities whose names passed in arguments must be present in the composite entity in which figure the rule. EXCLUDED

`the rule`
EXCLUDED `expresses` which if one of the entities whose name passed in argument, is present, the entities corresponding to the other arguments must miss in the made up entity in which figure the rule. Otherwise-known as if several entities of the group are present the rule is violated. TOGETHER

`the rule`
TOGETHER `then` expresses that if one of the entities whose name passed in argument is present in the composed entity, all those corresponding to the other names will owe the being too. The order of the occurrences does not have importance. And if none the entities represented in the rule is present in the composite entity, the composite entity are valid. PRESENT

`_PRESENT rule`
PRESENT `_PRESENT expresses` that if the entity corresponding to the first **name is** present, then all those corresponding to the other names will also owe the being in the current composite entity. The order of occurrence of the other entities does not have importance. If none the entities represented in the rule is present, the composite entity are valid. PRESENT

_ABSENT rule
PRESENT _ABSENT expresses that if the entity corresponding to the first **name is** present, then all those corresponding to the other names will have to miss in the current composite entity. The order of occurrence of the other entities does not have importance. If none the entities represented in the rule is present, the composite entity are valid. Each

rule of composition (called also simply “rule”) is a class (see the modulus regle.py). Taken in charge of

1.2 maintenance the following

approach is proposed to the candidate with the maintenance of the Eficas graphic interface and/or the supervisor of Code_Aster . *To study*

1) “Accas” i.e. what is common to the Eficas graphic interface and the supervisor of Code_Aster *; To study*
2) structure of the general catalog of the commands: in the file catalogues and the memory of the application. For that: to familiarize itself

•with the techniques of programming in Python, used in Accas; to develop
•a small model of factory (cf [§4.1]) for4.1 well the basic mechanism of the loading of the key words. To study

1) structure of the command set (in its file) and memory; in particular question of the loading of the command set (mechanism and zones of codes concerned) must be considered; To examine
2) scripts Python or the sources C concerned at the time of requests for modification or processing of the errors detected by the users. The organization

2 sources Python Conventions

2.1 following

conventions the purpose of which are facilitating the reading of scripts, are imperfectly applied a name of

1) class starts with a capital letter; the identifier
2) of an object of the type Python list starts with the prefix l_ (this rule is used in Eficas); in
3) the packages used by the supervisor (Core, Execution , Validation , Build and Accas) only one class is defined by modulus i.e. by file *.py; in
4) the packages used by the supervisor (Core, Execution , Validation , Build and Accas) the name of each modulus starts with a prefix indicating the name of the package. N_ for
    1) V_ Core for
    2) E_ Validation for
    3) B_ Execution for
    4) Build A_ for
    5) Accas Typology

2.2 of the moduli Python Each class

is defined in a modulus: for example, class MCSIMP is `defined` in the N_MCSIMP.py modulus `where the N_` prefix indicates the name of the package (Core) containing the modulus. The technique

of the packages makes it possible to cut out the moduli according to the sphere of activity in which it acts. A each field corresponds a package Python For example

, class MCSIMP exists in each of the five packages, ·Core:

1) N_MCIMP.SIMP ; ·`Validation`
2) : V_MCSIMP.SIMP ; ·`Ihm`: I_
3) MCIMP.SIMP ; ·`Accas`:
4) A_MCIMP.SIMP ; ·`Build`:
5) B_MCIMP.SIMP ; `Core This` package

contains primarily the system of class of the factory of the command set. Validation

`This package`
contains the moduli carrying out the checks of validity of the objects (conditional commands, blocks, key words, …). Build This package

is present only in the supervisor contains the moduli treating the commands of the macro type and the methods of request with the command set since the API-C: the GETVxxx interfaces. Accas This package

is most important. It contains – in particular - the classes more including used as well by the interface – Eficas graph as by the supervisor of Code_Aster. *It is in* this package that the objects should be searched and the methods specialized in the processing – not graph – commands: loading

1) of the catalog; loading
2) of a command set; execution
3) of the command set. The classes

girls defined in this package are it by heritage of classes relationships having the same name as the classes girls but being located in a different package. Ihm the classes

of this package enrich the classes by the Core of methods – nonrelated to the graphic aspect – used by the Eficas graphic interface. Editor This

package
contains the moduli of graphic treatment of the command set. Hierarchy

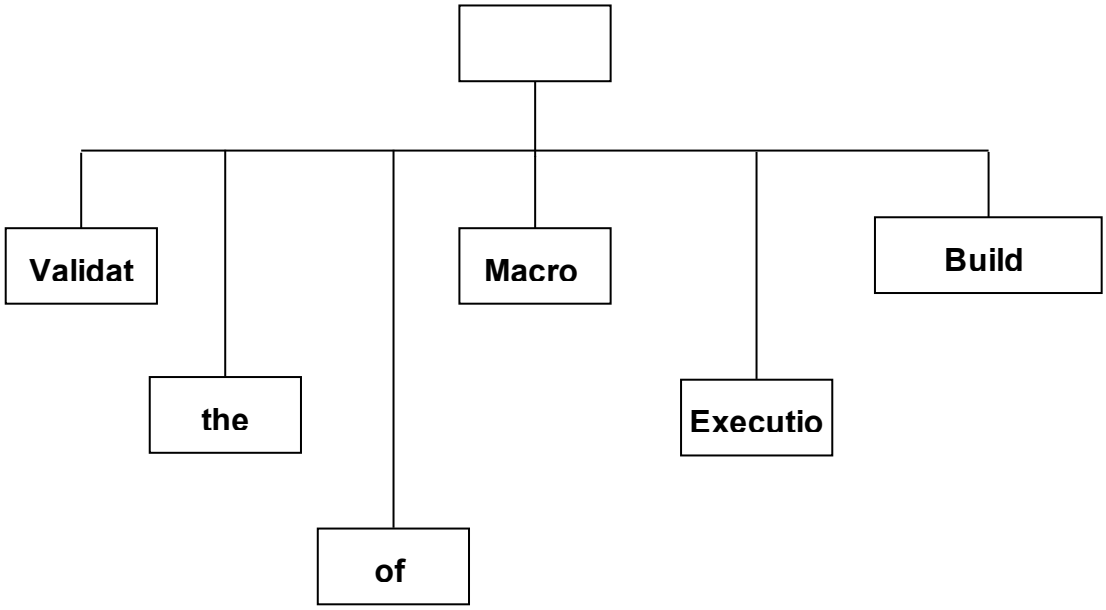
2.3 of the directories of the sources the python

2.3.1 sources of the supervisor the supervisor

of Code_Aster *is composed* of written moduli python: in C for L

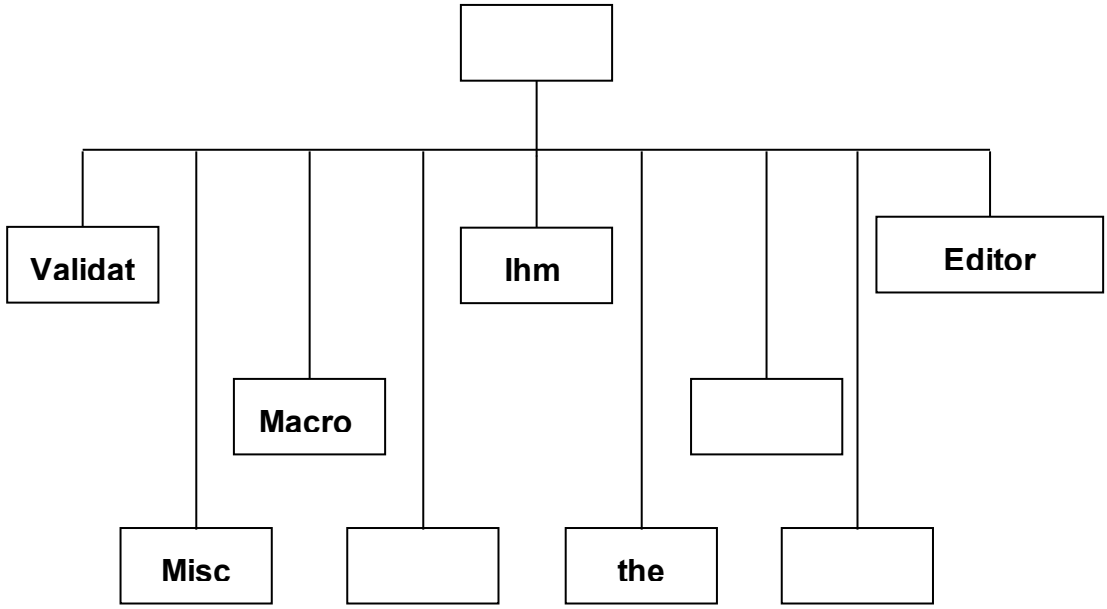
1) "interfaces applicative of the supervisor: the modulus aster (astermodule.c); in Python for
2) the management of the command set (loading, organization,…). The diagram below

presents the directories containing the Python sources of the supervisor. Core Eficas Accas



2.3.2 L" Eficas interface-graph the sources

are organized in directories under the directory of Eficas installation: Core `Eficas` Accas



2.4 This paragraph

describes the pre-necessary conditions with the operation of the Supervisor and the Eficas graphic interface. Installation of

2.4.1 the interpreter Python A to inform Shell

Parameters

2.4.2 of configuration A to inform Installation

of

2.4.3 the Eficas interface-graph A to inform Updated

of

2.4.4 the supervisor A to inform Use

of

3 the language Python in Eficas and the supervisor of Code_Aster One *presents here*

the techniques of programming in language Python, whose control is necessary to any candidate with the maintenance of the graphic interface Eficas and the supervisor of Code_Aster. Call *of*

3.1 a function with a variable number of arguments the end-user

of Code_Aster uses *a script* Python to start the features and to provide values to the parameters of the functionality. The supply of these values being sometimes optional, the Accas core uses the mechanism envisaged in the language Python to pass a variable number of arguments to a function. To maintain

Accas, it is, consequently, necessary to control this technique whose we present a small example below. # script main.py

```
def fonc (number
, *tup_arguments, ** d_arguments): print number print
    repr (tup_args
    ) print repr (d_args
    ) fonc (11111, "arg 2

", "arg 3", 4, n=5, j=6) Under Unix, the interpretation
```

of script main.py is done by : `$ python` main.py It give

```
result
```

according to on the standard output: 11111 ("arg 2", "arg 3"

```
, 4) {
": 5, "I: 6} Only L
"argument number
```

is compulsory. The possible positional arguments according to are stored in a tuple (which can be empty) and the possible arguments passed by key word are stored in a dictionary. This technique is used

in particular, by the objects which build the command set in the memory then which L" initialize. Use of spaces

3.2 of name Notion of space of names

3.2.1 a space of names (see

[feeding-bottle *1], page 97)* is a dictionary Python containing *a set of* couples name/value. The name is in general a character string and the value can be a numerical value, a function or an object. In a modulus Python,

each instruction is carried out in a specific space of names called local space of names whose **contents can be** displayed by the function locals (). The instructions `also have` access to the total space of names whose **contents can be** displayed with the function globals (). Accas uses `in particular`

, a space of names to store the dictionary of the definitions which will be used to interpret the key words and to charge their value in the memory. The modulus __builtin

3.2.2 a remarkable modulus is

the standard modulus __builtin which – except typical case ([bib2], page 100) – is referred in each modulus user by the attribute __builtins in ReadOnly mode only **. The interpretation of**

the following sequence: # script main.py print globals

```
() poster {"
__doc": Nun, "_
```

_name"

```
: "__main", "__builtins": <module'__builtin "(built-in) >} the modulus __builtin
```

is imported by default ; all the data and the functions qu" it contains are thus accessible by defaults, in all the moduli of the application. The data and the functions defined in the modulus __builtin can thus be regarded as most total with the application. Inter alia, one finds in

this modulus, the following tools: the functions locals (). and

1) globals (); the variable _ _debug which
2) conditions interpretation according to its value 1 (default) or 0; the function "builtin"
3) __import. # script main.py # One installs

```
the modulus context
in the total space of interpreter # (__builtin) # under name
CONTEXT in order to
have access to functions # get_current_step, set_current
_step and unset_current_step of anywhere importation context importation
__builtin
____builtin_
_.CONTEXT=context It
is possible to import
```

and enrich the modulus __builtin; this technique is used in Eficas in Noyau/__init__.py script. Execution of a command Python

3.2.3 in a space of names In the following example,

variable GB defined at the beginning of script is accessible in all the units from the modulus: it is definite within the space of names total. # script main.py gb=2 def fonc (a)

```
: b=gb*a print
"locals
() =", locals
    () print
    "globals () =", globals ()
    return B x=123 z=-1 u=fonc (X) print
    z+u L
"interpretation

of
script main.py
```

poster: locals () = {“B”: 246, “has” : 123} globals

```
() = {” __doc “: Nun, “fonc”:
<function fonc At 0x810aee4>, “Z”: -1, “X”: 123,” __builtins “: <module” __builtin
“(built-in) >,” __name “: ” __main “, “GB”: 2} a specific space of names can
```

be created and used to carry out instructions stored in a character string. # script main.py d_contexte= {” a'

```
: 1, “B”:
2} print d_contexte s_commande=' x=a+b' exec
s_commande in
d_contexte print d_
context When he interprets
script main.py
```

, the interpreter enriches space by names d_contexte with “X”: 3 result of the instruction s_commande . But the local space of names does not contain result the instruction s_commande. During the use of L “instruction

exec, one can specify that the data created owe to L” local being within the space of names as the example shows it below. # script main.py # script main.py d_contexte= {

```
“has”: 1, “B”: 2
} print d_contexte
s_commande=' x=a+b' exec s_commande
in d_contexte
, locals () print “
x=”, X what displays: {“B”: 2, “has”: 1
} x= 3 the instruction
```

Python exec

```
makes it possible behind the key word
to create
```

a space of names by the interpretation of a script Python - stored in a character string - by specifying a space of names in, for example: exec s_script in space a second way import a modulus

3.3 the function “builtin” __import ([bib2]

, p. 100), makes it possible to import a modulus from its name stored in a character string. In fact, this function is called by the interpreter Python at the time of the importation of a modulus by the instruction importation ([bib3], p629). This technique is used on several occasions in Eficas (packages Editor and Extensions). nom_module=' string' print to dir () modulus _string =

```
__import (nom_module
) print to dir
() print to dir (module_string) print modulus
_string.lower
(“ABCD”) One notices that
the function __import turns over in
```

module_string, a reference on the modulus G-string from which it is possible to reach the tools contained in the modulus. The function “builtin” __import can also

be used to import a specific modulus from a parcelling (package). The following example imports the general catalog of the commands of Code_Aster then poster on the standard output *the name of each* command. importation sys TOP='/local/yessayan/Eficas/EficasPourSalome

```
"# directory
D" sys.path.append installation (TOP) sys.path.append (TOP+'/Aster') from
Catastrophes importation catastrophes for
command in cata.JdC.commandes: print commande.nom
It can be written:
importation sys TOP='/local
/yessayan/Eficas/EficasPourSalome' sys.path.append (
    TOP) sys.path.append (TOP+'/Aster
```

") package=__import (

```
"Cata.cata")
module_cata= getattr (package, "catastrophes") for
command in module_cata.JdC.commandes:
print commande.nom Thus qu" it
is indicated in the documentation of
python (http://www.python.org/doc/current/lib/built-in-funcs.html)
, the instruction
    __import ("Cata.cata
```

") imports the package Catastrophes. It thus remains to recover the modulus of the catalog starting from the package for example: module=getattr (package

, "catastrophes")) The command

__import makes it possible to import

a modulus or a package whose name is known only at the time of the interpretation

of current script. Modulate

Python written in language A.C. give the general catalog of the commands of Code_Aster Example 1: a factory to build the simple key words

3.4 Principle of the factory the end-user

4 of Code_Aster provides values to *the features*

4.1 via key words. A key word makes it possible

4.1.1 to introduce a value

(key word simple) or another *key word* (factor key word). The mechanism of loading and especially of use of the value of the key word is complicated by the fact that the type of this value is not single. This

type can be for example: float, int, G-string,… In the following example: MASSE_VOLUMIQUE=7800.0 FICHIER_MAILLAGE=' maillage.unv' key word simple MASSE_VOLUMIQUE and FICHIER_MAILLAGE receive value 7800.0 respectively

```
and "maillage.unv" It is thus necessary
to describe some share the characteristics
```

of the simple key word (name , standard , value by default , unit, optional or compulsory statute ,…). The response

with this question is founded on the separation characteristics/following value: the main role of an object of the type MCSIMP is to wrap the value of a key word the simple otherwise-known as principal attribute of an object of the type MCSIMP

1) is a value; an object of the type SIMP plays two leading roles: **to wrap the definition of** a simple key word: an object SIMP contains all the characteristics of
2) a simple key word : its name, the type of its value

•,…; but this object also has a function __call which makes it possible to generate a simple key word from its characteristics; an object SIMP can be regarded
•as a machine to manufacture an object of MCSIMP from where the term factory Each object MCSIMP contains its value and a reference on L "object SIMP which describes its characteristics. The technique of the factory is presented *Ci below*

by reducing classes MCSIMP and SIMP to their minimum. Class MCSIMP has two attributes: definition

: its definition (a reference on L" object SIMP which created object MCSIMP) from which it is

1) possible to recover the text part

•of the key word : definition.nom or its definition.type type; val : its value class SIMP has two attributes: name: the part text (character string which can contain a white space) key word MCSIMP which object SIMP
• will build

1) ; type: the type of the value by the key word
   1) introduces MCSIMP which object SIMP will build. The function __call of class SIMP creates an object of the type MCSIMP.
   2) That implies that class MCSIMP is defined before the definition of this function . In the model

of design factory applied to the set of commands of Code_Aster , the class manufactured (example MCSIMP) must be defined before the class producer

```
(SIMP). # script main.py importation sys # G-string in python 2.1 becomes str from python 2.2
s_version=str (sys.version_ information [0]) + '. "+str (sys.version_info [1]) assert (float
(s_version) <=2.1
```

```
), "the version "+s
_version+"
of python is INVALID" class MCSIMP: def __init (coil,
definition, val, parent=None): self.definition = self.val definition
= val self.parent = Nun class SIMP: def __init (coil, name, type): self.nom
=

name self.type =type
    def __call (coil, val, parent=None): assert (
        str (standard (val))== " <type" "+self.type
        +" ">"        ) return
        MCSIMP (=self

definition, val=val
    , parent=parent) d_context= {"MASSE
        _VOLUMIQUE"
        : SIMP (VOLUMINAL nom='
    MASSE", type=' float'), "FICHIER_MAILLAGE
        ": SIMP (nom=' FICHIER MAILLAGE', type=' string')
        } s_commande = "rho=MASSE_VOLUMIQUE (7800.0)" exec s_commande

in d_context
        , locals () # rho  is added to the space of names locals () print
                rho  print  rho.definition.nom  print  rho.val  s_commande  =
"mail=FICHIER_MAILLAGE (
"maillage.d

")" exec s_commande in d_context, locals
() # mail is added to the space of names     locals  ()  print  mail  print
mail.definition.nom print mail.val sys.stderr.write ("
FIN NORM
of main.py " + ' \) the interpretation
of


script main.py above gives the following display
: <__main__.MCSIMP instance VOLUMINAL At       0x810c4ac>    MASSE    7800.0
<__main__.MCSIMP instance
At 0
x810c4d4> File mesh
maillage.d Let us consider

the lines s_commande = "rho=MASSE_VOLUMIQUE
```

(7800.0)" exec s_commande in `d_context` , locals () the mechanism of construction

```
of the rho object, within the space of
names room, is
a mechanism
of factory including the following
stages: Within the space of
```

names d_context,

```
the command becomes: rho=SIMP (VOLUMINAL nom=' MASSE
", type=' float") (7800.0) then it
```

is carried out; An object of the type SIMP is built with VOLUMINAL name “MASSE”; the method __call is called with val=7800.0 in

1) argument; starting from the name (self.nom) and value (val), the method __call creates and turns over an object of the type MCSIMP; the turned over object
1) is affected with the variable rho within the space of names room. Important
2) : In the command s_commande, the key word should not
3) contain of white space: ” rho=MASSE_VOLUMIQUE (7800.0) "is a valid instruction Python. A white space in
4) “rho=MASSE VOLUMINAL (7800.0)” would generate an error with interpretation

. In the dictionary

of the key words the character strings used for the words, must obey the writing rules of an identifier Python: no

white space. The organization of the factory in files the organization in files, presented

below, described that which is used for the Eficas graphic interface and the supervisor. It also describes the procedure of loading of the command set in the memory

4.1.2 starting from the furnished information –

in the command file - by the end-user. The files all – scripts Python - must be defined in the following order: MCSIMP. py SIMP.py dictio.py: the dictionary of key words MASSE_VOLUMIQUE and FICHIER_MAILLAGE valeurs.py: the file of the values provided by the end-user (for Aster, the command file

user) main.py: the code charging the values provided by the end-user

•
•by reading
•and interpreting the file valeurs.py (for Aster the executable one , also interpreter
•python : aster.exe) Let us recall that the definition of class MCSIMP must be carried out before that of class
•SIMP . What leads to an organization starting with the definition of class MCSIMP. # script MCSIMP.py class MCSIMP: def __init (coil, definition, val, parent=None): assert (definition

. __class. __name__==' SIMP') self.definition = assert definition (standard (val). __name__==definition.type) self.val = val self.parent =None return the attribute self.parent useless here, will be used when

```
the key word is

defined inside
    a command. It will then contain a reference

        on this command. The developer
        of Accas can then define

        class SIMP. # script SIMP.py importation MCSIMP
        importation types
        class SIMP: def
```

```
__init
```

(coil, name , type) : self.nom = name self.type =type def __call (coil, val, parent=None): assert (str (standard (val))== " <type ""+self.type+" ">") return MCSIMP.MCSIMP (definition=self

, val=val, parent=parent) Once two moduli SIMP and MCSIMP

```
placed at its disposal

,
the developer

of numerical
    features of the code can now
        define
        a "application
    catalog of key words" in the file
        dictio.py. This script defines in a dictionary
        python, description (standard, possible values, field of definition
```

,...) values associated with the key word with kind to allow the reading of these values. For example: # script dictio.py from SIMP importation SIMP dict= {"MASSE_VOLUMIQUE": SIMP (nom=' MASSE VOLUMINAL", standard = ' float', "FICHIER_MAILLAGE": SIMP (nom= `FICHIER MAILLAGE', type=' string')} And the end-user can finally use the features by providing a script, for example valeurs.py. # script valeurs.py rho=

MASSE_VOLUMIQUE (

```
7800.0) mail=FICHIER

_MAILLAGE ("maillage.d
") to charge in memory the abundant data by the user
          , the code will read the command file as follows: # script
```

main.py from SIMP importation * d_context= {"MASSE_VOLUMIQUE": SIMP (nom=' MASSE VOLUMIQUE', type=' float'), "FICHIER_ MAILLAGE" : SIMP

```
should be interpreted (

nom=' File mesh ", type=' string")
} nom_script _valeurs = "valeurs.py" f=open
```

(nom_script _valeurs, “R”) string_parametres = f.read () f.close () exec string_parametres in d_context, locals () print rho, rho.definition.nom,

```
rho.val to charge

in memory,


the value associated with a simple key word, script python -
               text of the key word simple and value (S) associated (S)
provided by


the end-user - within the space of
names (the dictionary d_
context) of the key word. Example
2:

a factory to build a command In practice

, the key words  are not separately
```

but obligatorily defined inside a command. What complicates the process of construction of the key words in the memory. We now present an example – always simplified – intended to facilitate the comprehension of this process

4.2 . For that we will consider that a command set

is a list of commands of type “procedure” and that each command is parameterized by key word simple and only by simple key words (not of conditional block, not of put-word factor). This simplification increases legibility while preserving all the categories of difficulties

of facing to charge the command set in memory. One thus starts by adding a factory of commands of the type PROC_ETAPE. Addition of classes PROC and PROC_ETAPE with the factory the class PROC_ETAPE which models an ordering of standard “procedure”, is very close to class MCSIMP. So much so that both could inherit a class common mother (it is the case in Accas besides). # script

PROC_ETAPE.py print “\ tImport of “+__name class PROC_ETAPE : def __init

4.2.1 __(coil, definition , arguments = {}): print 2*' \ t'+ " PROC

_ETAPE __ init: creation of an object "+ \ coil. __class. __name print 3*' \ t'+' PROC_ETAPE __init: definition.nom = ', definition.nom print 3*' \ t'+' PROC_ETAPE __init: arguments=', arguments assert (definition. __class. __name__==' PROC') self.definition

```
= definition self.valor

= arguments return an object PROC_

STAGE is manufactured
    by an object of the type PROC. Its attribute self.definition
        is a reference on the object PROC which created it
        . # script PROC.py print
             " \ tImport of "+__name importation PROC_ETAPE from SIMP
importation * class
        PROC: def __init (coil, name, op, ** arguments

        ): print 1*' \ t'+ " PROC __init: creation of
        an object "+ \ coil. __class. __
        name print 2*' \ you

        + ' PROC
```

__init: nom=', name print 2*' \ t'+' PROC __init: arguments=' , arguments self.nom = name # text of the command self.entites = arguments # dictionary of the manufacturers self.op

```
= op # number of
operator FORTRAN return def
__call (coil
, ** arguments): # arguments

contains
    the definition of the values of MCSIMP (MASSE_VOLUMIQUE
        , # FICHIER_MAILLAGE) print 1*' \ t'+' PROC __call
        __: arguments=', arguments print
        1*' \ t'+' PROC __call: self.entites=', self.entites
        # construction of the simple key words
        of the command          and addition in # the dictionary
        of key words     of the command PROC in the course of
        # construction            dict = {} for K, v in args.items
        ():

    dict [K] = self.entites [K] (val=v
         , parent=self) print 1*' \ t'+' PROC __call: dict=', dict return
PROC_
        ETAPE.PROC_ETAPE (coil
        , dict) During its creation carried out
        starting from the catalog, object PROC memorizes in
           the dictionary self.entites, the composition of the command in
simple
        key words; this information will be used in the second
        time -
        to build
        the simple key words of the command
            - when object PROC is called upon
        via its method __call. It is also
        the method __call which will initialize
```

the key word located inside the command with the values provided in the command file (modulus SIMP is exactly that presented in the first example). Notion of catalog of commands In the first example ([§4.1]), the description of the key words was made in the dictionary (L "spaces names) d_context. But

to facilitate the task of the developers of Code_Aster, it is preferable to describe the commands and their contents via a script Python then to convert this script into a dictionary which will serve D" space names

4.2.2 for the loading of the commands.

For our second example,4.1 can be written thus; # script cata.py print 1*' \ t'+ " Importation of "+__name from SIMP importation * from PROC importation * AFFE_MATERIAU=PROC (nom=' AFFE *_MATERIAU',* op=10, MASSE_VOLUMIQUE=SIMP (VOLUMINAL nom= " MASSE", type=' float'), FICHIER_MAILLAGE=SIMP (nom= " File mesh", type=' string')) This catalog contains only one command: AFFE_MATERIAU of which the use will start

the call to routine FORTRAN op0010. This routine will use two

```
parameters MASSE

_VOLUMIQUE and FICHIER_MAILLAGE

the conversion of the catalog
into a dictionary

is done into important simply
                    the catalog
                    in a space of names. What does the following sequence
                     : d_context= {} string_cata= " from catastrophes importation *"
exec G-string
```

_catastrophes in d_context Notion of command file the command file is also a script Python, very simple intended to be interpreted within the space of names of the catalog of commands. For example:

script commandes.py AFFE_MATERIAU (MASSE_VOLUMIQUE=7800.0, FICHIER_MAILLAGE= " maillage.unv") In script commandes.py above, the end-user

```
requires
the execution of routine FORTRAN
op0010 with MASSE_VOLUMIQUE
```

4.2.3 =7800.0 and FICHIER_MAILLAGE

= " maillage.unv". He is interpreted by the following sequence: f_commandes=open ("commandes.py", "R") string_commandes = f_commandes.read () f_commandes.close () exec string

```
_commandes in d_context

At the end of which the command set (here reduced to only command
```

AFFE_MATERIAU) is defined within the space of names d_context. Use of the catalog and the command file script following Python carries out the loading of the catalog, the loading of the command file and the examination

```
of the command set in the memory. #
script main.py importation traceback trace=traceback.extract
_stack ()
script_file=trace [0] [0] prefixe=script
```

_file+': “print prefixe+ " debut of “, script_file d_context= {} # 1. Loading of catalog # Creation - into important the catalog catastrophes

4.2.4 - of a space of name serving # for interpretation

of the command set print 3*” \ n'+prefixe+ " importation of the catalog” string_cata= " from catastrophes importation *” exec string_cata in d_context # 2. Loading of the text

```
of commands #
Reading of the command file
(the text of the commands
east stores in
one # chains caratères
) print 3*' \ n'+prefixe+ " reading of the text

of the commands
” f_commandes=open (“commandes.py
”, “R”) string_commandes = f_commandes.read () f_commandes.close () # 3.
Creation
of the command set # Interpretation

of the text of the command in the d_contexte of
the catalog. Structure # command set
, produced, is stored

within the space of name d_context print
3*' \ n'+prefixe+ \ “Conversion of the text of the commands (G-string) into
a command set
(d_context) “exec

string_commandes in d_context # 4. Path of the structure
command set in the d_contexte
print 3*' \ n'+prefixe+ " Display of
the standard command set”

importation for K, v in d_context.items
(): standard yew (v) ==types.InstanceType and v. __class. __name__==
“PROC_ETAPE”: # if
the attribute is a command, one examines his value # it be-A-to say its key
words
print 1*” \ t'+v.definition.nom+' \
t'+str (v. __class) for kk, vv in v.valeur.items (): print 2*” \ t'+kk, “:
”, vv, “\ you, vv.val print 2*'

\ print prefixe+ " FIN NORM of”,

script_file the file catalogue Catalog of a command the catalog

of a command contains description of the command
```

```
. Each
command is instance of
         the class OPER, PROC or MACRO Attribute Description name name of
the command
                (character string without white space) op number
                of operator FORTRAN:
                    integer ranging between 1 and 199 sd_prod standard of
result
                , for the commands of the type
                        OPER regulates list of the rules of composition

of the command
Fr comment in French Doc. reference
```

4.3 of documentation

4.3.1 Aster reentrant repetable

entities Composition of the command: arguments containing the description of the key words used for of the command providing values

to	the key word
=maillage .03-e', , min catalog	Example ASSE_MAILLAGE=OPER (nom=' ASSE_MAILLAGE', op= 105, sd_prod , fr=' To assemble two meshes under only one nom', docu=' U4.23 reentrant=', MAILLAGE =SIMP (statut=' o', typ=maillage =2, max=2),); In this example, drawn from the true of commands of Code_Aster : the described command is called MAILLAGE

ASSE_;

it makes it possible	to start operator FORTRAN op0105; it turns over a data of mesh type; this type is defined in the beginning of catalog

cata.py; L

```
"use of L" operator FORTRAN op0105 requires obligatorily
                  , the supply of 2 data of mesh type
                  general Structure of the file
         catalogues        the file catalogues commands of Code_Aster
```

– the modulus $TOP/Aster/Catastrophes/cata.py - contains following information

1) : importation of all information of the Accas
2) modulus, in particular Accas.A_ASSD.ASSD declaration of
3) the types deriving from the generic type Accas . A_ASSD.ASSD, used to typify the values of the key words
4) or the values turned over by the commands ; for example types: integer, reality, complex, list, character string

4.3.2 ; the geometrical ones, No (node), groupno

, my (mesh), groupma; mesh, *model, MATER* etc the list of the catalogs of the commands i.e. the description of

1) all the commands, with for each command Installation of the catalog in the memory
2) It is important to remember that a reference on the catalog running is stored in the modulus whose reference is stored in CONTEXT. Reference CONTEXT is itself

•definite within the space of names total
•__builtin a reference on the catalog running is obtained by
•: CONTEXT.get _current _catastrophes
•*()*

1) the data structure in the package OBJECTget_val Core (): type_de_baseget_etape (): ETAPEMCSIMPget_val (): type_de_baseJDCget

4.4 _etape (): ETAPEMCFACTget_val (): OBJECTMCBLOCget

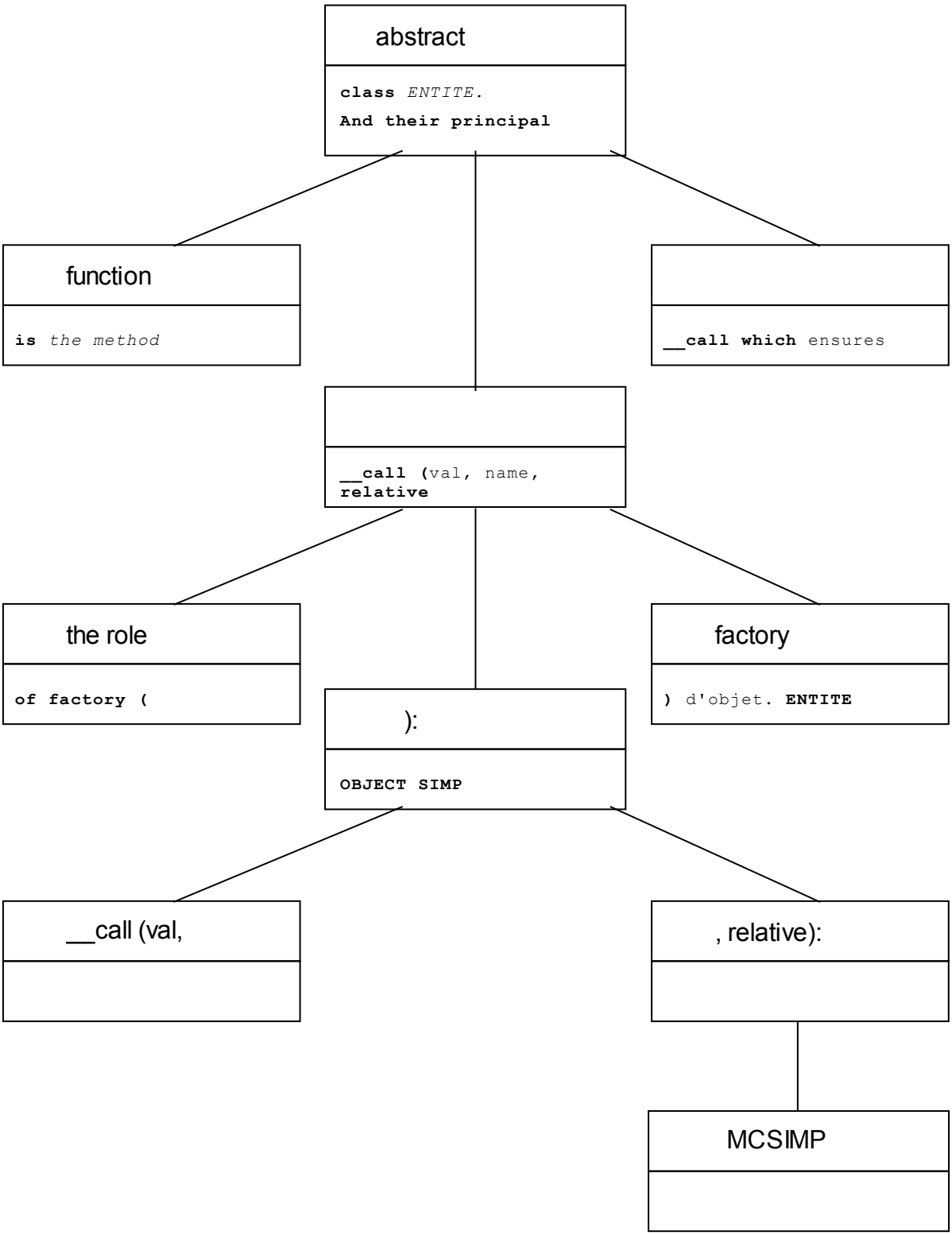
_val (): OBJECTMCCOMPOget_mocle (key): OBJECTETAPEget_etape (): ETAPEPROC_ETAPEMACRO_ETAPEFORM_ETAPELes following classes have as a role to store the command set in the memory and to restore the value of the key words to the request.

All the classes presented below take part in

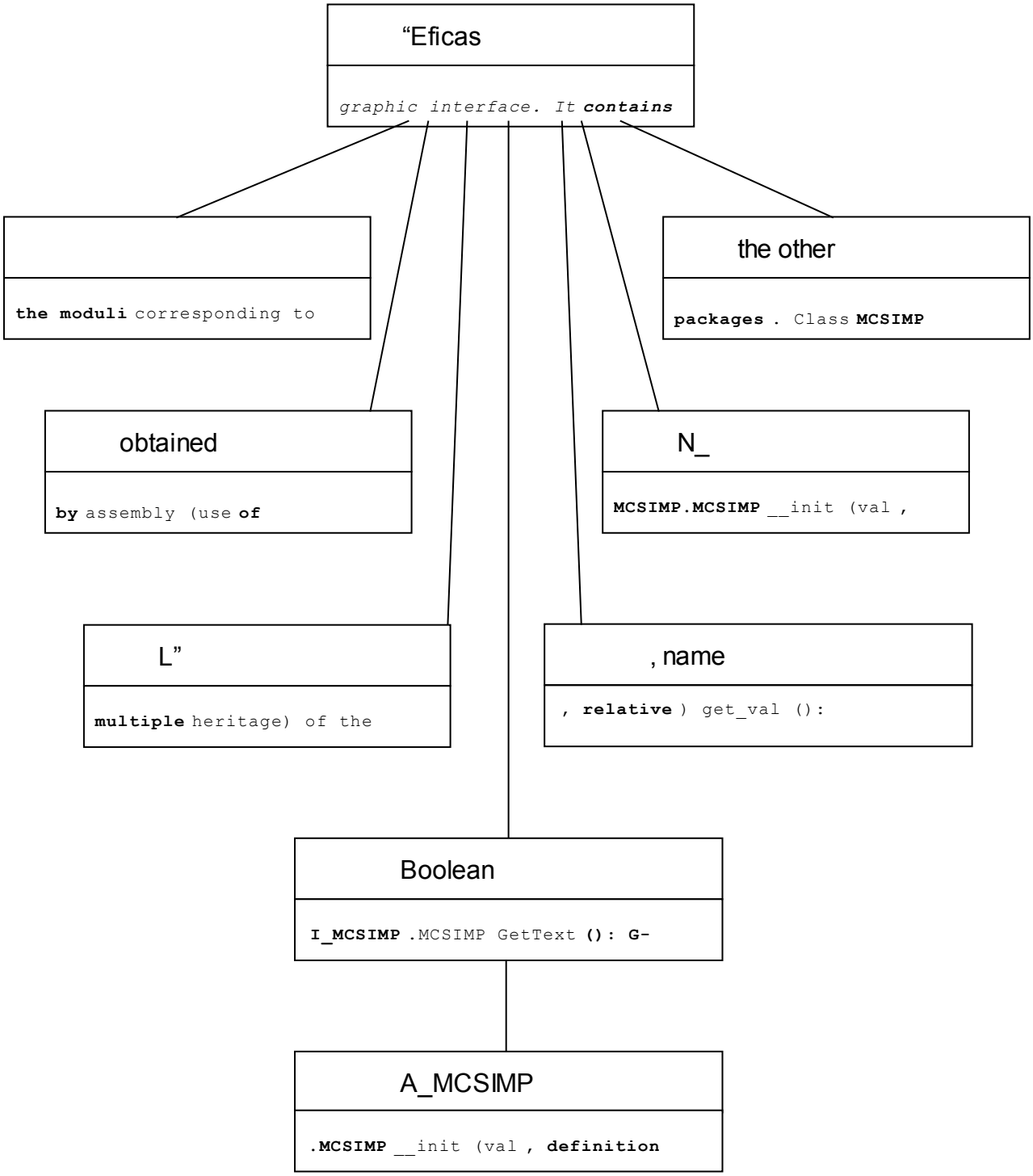
the loading of the command set

4.4.1 in the memory. They inherit all

__call (val, name, relative): MCFACT BLOCK __call (val, name, relative): MCBLOC PROC __call (** arguments): PROC_ETAPE OPER __call (reuse, ** arguments)



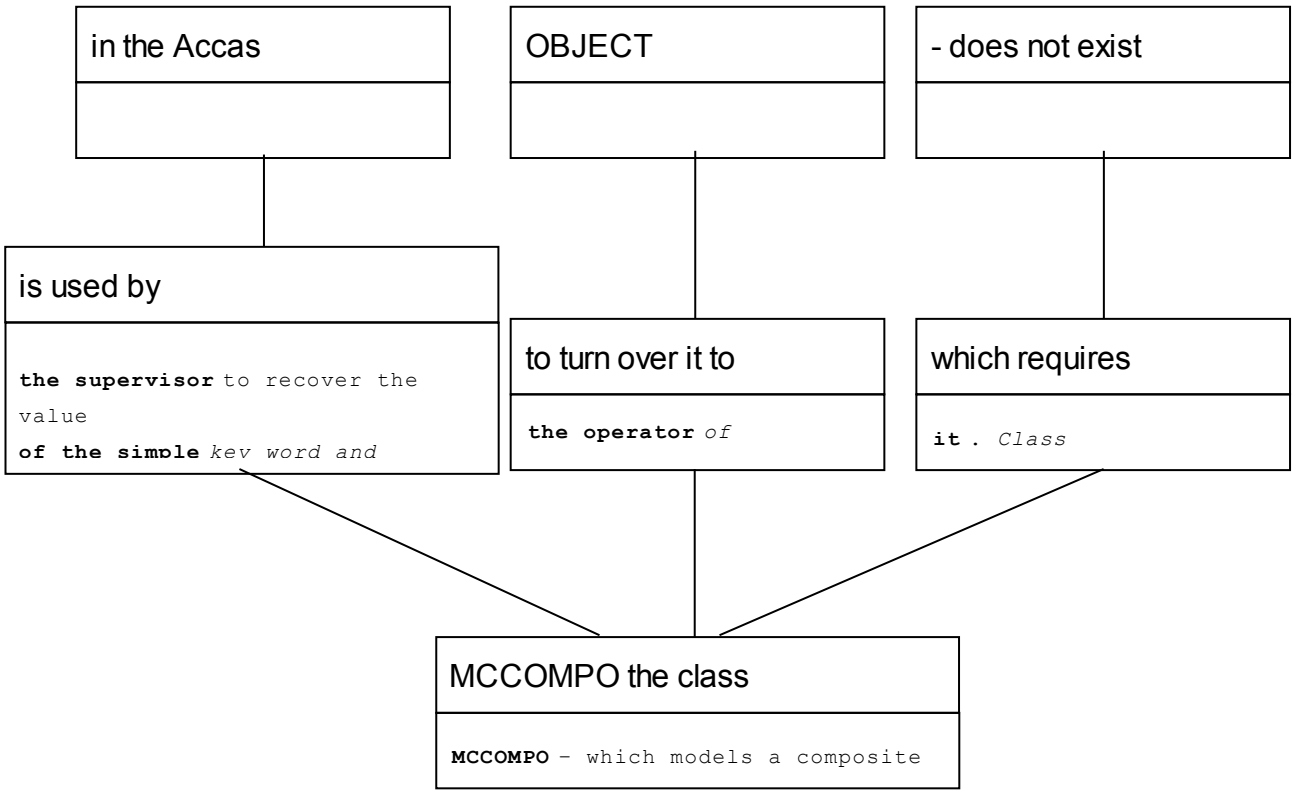
: STAGE JDC_CATA __call (procedure, catastrophes, catastrophes_ord_dico, name, relative, ** arguments): JDC MACRO __call (reuse, ** arguments): MACRO_ETAPE FORMS __call (reuse, ** arguments): FORM_ STAGE the package Accas the Accas package is the principal package used by the supervisor of Code_Aster and L



4.4.2 , name, relative) V_OBJECT

.OBJECT I_OBJECT.OBJECT N_OBJECT.OBJECT the objects of class MCSIMP *are built* with the manufacturer of class A_MCSIMP.MCSIMP. The diagram above represents the principal contribution of each package to the features of class MCSIMP the method

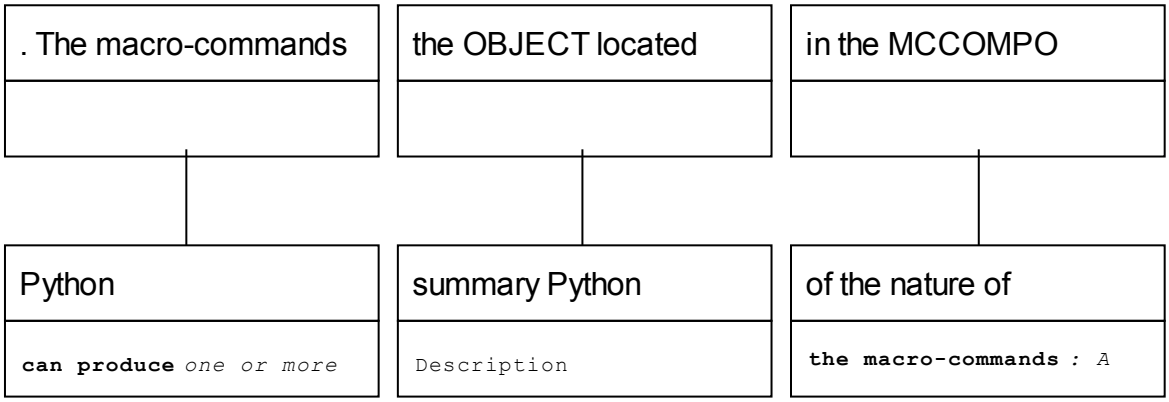
get_val ()



package but it is important because it is used as a basis for classes MCFACT, MCBLOC. V_OBJECT.OBJECT I_ OBJECT. OBJECT N_OBJECT.OBJECT V_MCCOMPO.MCCOMPO carryforward (): G-string I_MCCOMPO. MCSIMP getlabeltext (): G-string N_MCCOMPO .MCCOMPO build_mc ( ): type_de_base the diagram above highlights behaviors justifying the existence of class MCCOMPO *. the method* build_mc ()

in the package `Core`

: it `builds` the objects located inside the OBJECT; the method carryforward () which turns over the ratio of validation by applying the method `isvalid () to all`



results (called concepts) whereas the simple commands produce zero (ordering `of type`

1) PROC) or one result `(ordering` of type OPER); A macro-command has parameters like an ordinary command ;
2) they are simple keywords factors and key words; The principal product concept by macro `is turned over` by the macro one `while`

5 the secondary product concepts

are arguments modified by the macro one;

1) The secondary product concepts must be typified: that is done via a function provided by the developer of macro via the argument sd_prod
2) the macro one; The body of the macro-command is a function fascinating Python charges with it the processing which includes the call to other
3) commands (or even with other macro-commands). To define a macro-command, its developer must thus define: key words of the command;
4) the type of the product concepts; the body of the macro-command. Is some questions How of knowing which file catalog used by a computation? The situation is the following one: a computation was carried out with
5) Code_Aster; several files of catalog of the commands are present in the environment; the user – or the developer – wants to know that which

is actually used. A solution can be as follows: to import the catalog

1) in the command file
2) , for example in ahlv100a.comm
3) ; to insert in the command file

6 the following

6.1 sequence which: import the catalog, writes the reference of the catalog

on the standard output,

1) stops the processing. importation catastrophes *, sys print*
2) “ahlv100a.comm: catastrophes=”, catastrophes sys.exit (0) With this sequence, one obtains result following
3) style: catastrophes=<module “catastrophes” from “/home/salome/yessayan/Devel/Asterv7/bibpyt/Catastrophes/cata.pyc

“> D” where L” one deduces that the catalog

1) used can be in the file: Is /home/salome/yessayan/Devel/Asterv7/bibpyt/Cata/cata.py

2) How mode DEBUG managed? In Eficas

•and the supervisor
•- in fact, in any script Python - mode DEBUG
•is managed via a definite

```
standard variable
within the space of names total _
_builtin:
```

__debug. In normal mode of interpretation (python main.py), __debug

```
__is put at 1 (in main.pyc) but in mode optimized (python - O main.py) __debug
```

__is put at 0 (in main.pyo) At any moment, in all the moduli, the variable

```
__debug can be used to condition the processing
```

6.2 . Where the catalog of command is

it charged in the memory That it is in the supervisor or the Eficas graphic interface, JdC, the Python object containing the catalog is created in modulus catastrophes of the package Catastrophes. More exactly, JdC is created at the time when the modulus catastrophes is imported : the importation is carried out in the method imports class SUPERV modulates Execution / E_SUPERV.

After the importation, the JdC object contains – in its attribute commands, of type lists - the definition of all

6.3 the commands available like that of all the key words

associated with each command. Creation of the JdC object At the beginning of script cata.py, JdC is declared by the instruction: JdC = JDC _CATASTROPHES (code=' ASTER', execmodul=None, rules = (AU_MOINS_UN ("debut", "POURSUITE "), AU_MOINS_UN ("FIN"), A_CLASSER (("debut", "POURSUITE"), "FIN"))) This instruction calls mainly on the method __init class N_JDC _CATASTROPHES .JDC_CATASTROPHES (package Core). In this method, the JdC object created is recorded in total space __builtins, via the variable _catastrophes in

6.3.1 modulus CONTEXT: __builtins

["CONTEXT"]. _catastrophes. A reference on the catalog running is always

```
available in L `total
               space of names
               __builtins. Loading of the entities
                         of the catalog in L
                         "JdC object After creation the loading
```

S" always carries out at the time of the importation of the catalog in the method imports, by creating objects of types OPER: PROC: MACRO: Where the command set is carried out by the supervisor ? The command set J (object of the Accas.A_JDC.JDC type) is carried out in the method Carries out class SUPERV

in modulus E_SUPERV of the package Execution. Two cases are possible: In mode PAR_LOT= "YES" (in

6.3.2 script the attribute j.par_lot of the command set is positioned

in "OUI"), the processing is carried out by the call j.exec_compile (); In mode PAR_LOT=' NON' (in script the attribute j.par_lot

1) of the command set
2)
3) is positioned

6.4 with "NON"), the processing is carried out by the call

ier= self.ParLotMixte (J). A what serves the key word _F used in the command file ? Into the command file , factor key word is introduced by

the character string

1) _F. In fact this character string is a name of class which to the key word deals with creation in memory of the dictionary corresponding factor from

a description

1) using the equal sign "=" and of the brackets rather than the two points ": " and the accodances which it would be necessary to use with a dictionary

standard Python. For example:

6.5 ELAS= _F (E = 2.1E 11 , NU = 0.3, ALPHA = 1.E-5, RHO = 8000

.) is equivalent to: ELAS= {E: 2.1E11, NU: 0.3, ALPHA: 1.E-5, RHO: 8000.} This presentation is more adapted to the wishes of the end-users and the tradition of the language of command of Code_Aster. Where the interface getvxx of the command set find-such? The methods getvxx belong to the class STAGE defined in modulus B_ETAPE of the Build package. Bibliography Introduction to Python, Lutz Mark & David Ascher

, O' REILLY
, Paris , 2001 Python, Essential Reference , David Mr. Beazley , New **Riders**
, 2001 Python 2.1
Bible, Dave Brueck & Stephen To tan, 2001

6.6

7

[1]
[2]

[3]