Maintenance du superviseur de *Code_Aster*

Résumé :

Ce document est à l'usage des développeurs du noyau superviseur d'Aster. Il explicite l'interprétation du fichier de données de l'utilisateur (construction du jeu de commandes) et l'enchaînement des exécutions.

Nota Bene important :

Ce document a été produit pour la version 6 de Code_Aster et n'a pas été mis à jour depuis. Il est conservé ici pour archive.

Table des matières

1 Introduction

Etude numérique

Pour réaliser une étude numérique avec *Code_Aster*, l'utilisateur final déclenche les fonctionnalités du code et fournit les informations nécessaires à l'exécution de ces fonctionnalités.

Une fonctionnalité est sélectionnée par l'intermédiaire d'une commande. Une commande est d'abord un nom c'est-à-dire une chaîne de caractère ou encore un identifiant **externe** (connu de l'utilisateur) de la fonctionnalité. Elle agrège également des attributs et en premier lieu l'identifiant **interne** de la fonctionnalité (exploitable par le code Fortran).

Les informations nécessaires au lancement et à l'exécution de la commande autrement-dit les paramètres du traitement numérique, sont des données introduites au moyen de mots-clés les identifiant. Pour une commande donnée, un certain nombre de paramètres doivent être définis.

La description des commandes et des mots-clés est effectuée par les développeurs de *Code_Aster*. dans un fichier appelé « catalogue des commandes ».

L'utilisateur final utilise le code via un fichier appelé « fichier de commande ». Il y fournit des commandes dont la composition doit être compatible avec leur description dans le catalogue des commandes. Les commandes sont nombreuses mais le nombre de données associées est beaucoup plus important avec des combinaisons possibles elles-mêmes très nombreuses.

Enfin le code de calcul stocke les informations (commandes et mots-clés) dans une structure de données interne appelée « jeu de commande ».

Le rôle du superviseur

Le superviseur est la partie de *Code_Aster* qui gère le jeu de commande. En particulier :

1) il importe le catalogue des commandes dans la mémoire Python,
2) il charge les commandes et leurs mots-clés dans la mémoire Python,
3) il exécute le traitement commande par commande,
4) il fournit - à la demande du Fortran – la valeur des paramètres aux fonctionnalités de *Code_Aster*. stockées dans la mémoire Python. Pour cela le superviseur propose une API C.

L'interface graphique Eficas

Il est possible de définir « manuellement » son fichier de commandes, par exemple au moyen d'un éditeur de texte. Toutefois, l'utilisateur non familier avec la syntaxe des commandes qu'il manipule mais aussi le langage python pourra utiliser Eficas.

L'interface graphique Eficas est destinée à l'utilisateur final de *Code_Aster*. Elle lui permet de construire des commandes valides avec des mots-clés associés eux aussi validés statiquement, puis de générer un fichier de commande à destination du code. L'utilisateur a l'assurance que le fichier produit par Eficas a une syntaxe correcte.

Le noyau

Les sources communes à l'interface-graphique Eficas et le superviseur de *Code_Aster* sont organisées (identifiées et regroupées) dans un espace de développement spécifique appelé le « Noyau ». La compréhension de ces sources est évidemment une condition nécessaire pour entreprendre la maintenance des deux outils qui les utilisent.

Les principaux acteurs

Les principaux types d'acteurs évoluant dans l'environnement d'Eficas et du superviseur sont listés ci-dessous.

1) L'utilisateur final : il connaît plutôt la partie physique du problème à résoudre et il a pour tâche d'introduire des données valides dans le code, de lancer des calculs et de dépouiller les résultats ; il peut utiliser l'interface graphique Eficas pour cela.
2) Le développeur de fonctionnalités dans *Code_Aster* : il pratique essentiellement la programmation Fortran des algorithmes numériques et, dans le cas d'une macro-commande, la rédaction du script Python correspondant ; il enrichit et/ou modifie le catalogue de commande et il utilise l'API C du superviseur pour récupérer dans son code Fortran, les valeurs fournies par l'utilisateur final.
3) Le développeur du superviseur dans *Code_Aster* : il rédige les scripts de lecture et d'interprétation du catalogue de commandes et du jeu de commandes ; il s'occupe également du module python `aster` écrit en C (`astermodule.c`) qui sert d'API au superviseur.
4) Le responsable de la maintenance du code, de l'environnement d'utilisation et de la gestion de configuration : il centralise les sources. En plus du code Fortran, ces sources comprennent celles du catalogue (Python) des commandes, les scripts (Python) du superviseur et celles du module `aster`.
5) Le développeur de l'interface graphique Eficas développe une interface homme-machine. Son rôle est de concevoir et de programmer des boîtes de dialogue ainsi que des enchaînements d'événement permettant la prise en compte des requêtes de l'utilisateur final et la vérification – à chaque instant - de la validité du jeu de commandes en cours de construction.

Dans ce document, on met plutôt l'accent sur les principes de la conception du noyau que sur le détail des scripts dont on ne peut économiser l'examen.

Sujets traités

Le premier chapitre propose une définition sommaire des termes utilisés abondamment dans la suite du document.

Le chapitre 2 présente les quelques conventions et l'organisation des fichiers sources en répertoires : les packages. Il décrit également l'environnement nécessaire au développement et au fonctionnement du superviseur de *Code_Aster* et de l'interface graphique Eficas.

Dans le chapitre 3, quelques rappels sont effectués sur le langage de script Python. Ils doivent orienter le futur responsable de la maintenance vers certaines techniques qu'il lui faudra maîtriser pour effectuer sa tâche.

L'important sujet de la factory de commande est évoqué au chapitre 4. C'est une base nécessaire à la compréhension de toute la structure du jeu de commande.

Enfin, une réponse est apportée au chapitre 5, à quelques questions dont on peut prévoir que tout futur responsable de la maintenance pourra se poser.

1.1 Définitions

Mode « par lot »

Le processus de traitement des commandes utilisateur par le *Code_Aster* peut s'effectuer suivant deux modes.

Dans le premier mode, le fichier commande est chargé en mémoire pour créer le jeu de commandes. Cette création du jdc permet de valider la syntaxe python (parenthèses, virgules), la syntaxe Aster (cohérence avec le catalogue) et de valider les concepts passés en argument. Enfin après cette vérification, le jeu de commande est parcouru pour effectuer les traitements numériques correspondants.
Ce premier mode est appelé « mode par lot ». L'utilisateur final sélectionne ce mode en spécifiant la valeur 'OUI' pour le mot-clé PAR_LOT sous la commande DEBUT.

Dans le second mode, le fichier de commande est chargé en mémoire pour créer le jeu de commandes. Puis les étapes (équivalentes aux commandes) sont construites et exécutées séquentiellement.
L'utilisateur final sélectionne ce mode en spécifiant la valeur 'NON' pour le mot-clé PAR_LOT sous la commande DEBUT.

Par défaut, le mode utilisée est PAR_LOT='OUI'.

Opérateur de *Code_Aster*

Un opérateur est une unité de *Code_Aster* prenant en charge une fonctionnalité du code. Concrètement c'est un sous-programme Fortran dont le nom est numéroté, par exemple la subroutine OP0001 qui charge un maillage dans la mémoire de l'application. La numérotation des opérateurs facilite l'association entre leur représentation interne (sous-programme Fortran) et leur représentation externe pour l'utilisateur final (commande).

Commande

Une commande est une chaîne de caractères identifiant un opérateur numérique. Elle permet donc à l'utilisateur final de déclencher l'exécution de cet opérateur à partir d'un fichier de données appelé « fichier de commande ».

Il existe 4 types de commandes : OPER, PROC, MACRO et FORMULE.

Le développeur de l'opérateur numérique définit - dans le catalogue de la commande - les caractéristiques de la commande et celles des mot-clés correspondant aux paramètres de l'opérateur numérique :

1) son nom (la chaîne de caractères utilisable par l'utilisateur final),
2) ses règles de composition en mots-clés,
3) un commentaire explicatif en anglais et/ou en français,
4) la clé définissant le manuel et le chapitre consacré à ce mot-clé dans la documentation de *Code_Aster*.

Commande OPER

En plus des attributs énumérés ci-dessus, une commande de type OPER possède l'attribut suivant :

1) Le type de la structure de données Aster produite par l'opérateur et retournée par la commande ;

Commande PROC

Une commande de type PROC a la caractéristique de ne pas retourner une valeur. Cette particularité mise à part, elle possède les mêmes attributs qu'une commande de type OPER.

Commande MACRO

Une macro est une fonction écrite en Python – par le développeur Aster - qui appelle des commandes – c'est-à-dire des opérateurs - de *Code_Aster*. Elle stocke des résultats qui pourront être récupérés via le superviseur.

Le texte de la macro peut être public ; dans ce cas il est stocké dans un fichier spécifique du sous-répertoire Macro. S'il est privé, il est placé ou importé dans le fichier de commande.

Une commande de type MACRO permet à l'utilisateur final d'utiliser la macro. Par exemple la commande `ASSEMBLAGE` permet de lancer la macro publique `assemblage_ops`.

Catalogue d'une commande

Le catalogue d'une commande est l'ensemble des instructions Python décrivant la définition de la commande c'est-à-dire les valeurs affectées aux attributs de la commande.

Le catalogue d'une commande est écrit par le développeur de l'opérateur numérique associé à la commande.

Catalogue général

Le catalogue général est un fichier Python contenant la description de toutes les commandes autrement-dit contenant les catalogues de toutes les commandes.

Jeu de commande

Le jeu de commande est la structure de données - organisée dans un objet Python – contenant l'ensemble des informations fournies par l'utilisateur final, dans le fichier de commande.

Fichier de commande

Le fichier de commande permet à l'utilisateur final de déclencher les opérateurs numériques portant les fonctionnalités de *Code_Aster* par l'intermédiaire des commandes.

Structure de donnée Aster

Une structure de données Aster est une organisation de données produites par un opérateur numérique de *Code_Aster* . Elle est identifiée par un type lui-même déclaré au début du catalogue (`cata.py`) ; ce qui permet de l'utiliser – symboliquement – dans le fichier de commande bien qu'elle soit produite par du Fortran.

Mot-clé simple

Un mot-clé simple est une chaîne de caractères identifiant une donnée utilisée en entrée par un opérateur (une fonctionnalité numérique de *Code_Aster*). Un mot-clé simple est donc défini à l'intérieur d'une commande de *Code_Aster*.

L'utilisateur final pourra fournir une **valeur** au paramètre d'une commande par l'intermédiaire du nom du mot-clé simple correspondant dans le fichier de commande.

Le développeur de fonctionnalité de *Code_Aster* définira quant à lui, les **caractéristiques** du mot-clé simple dans le catalogue de la commande contenant le mot-clé simple :

1) son nom (la chaîne de caractères utilisable par l'utilisateur final et par l'opérateur numérique),
2) le type du paramètre (entier, réel, texte, concept, …),
3) le statut du mot-clé simple (facultatif ou obligatoire),
4) la valeur par défaut à affecter au paramètre,
5) le nombre minimum de données que l'utilisateur final devra fournir derrière le mot-clé simple,
6) le nombre maximum de données que l'utilisateur final devra fournir derrière le mot-clé simple,
7) un commentaire explicatif en anglais et/ou en français.

Le superviseur de *Code_Aster* charge dans la mémoire de l'application, les caractéristiques du mot-clé simple, à partir du catalogue des commandes. Puis il charge (et vérifie) éventuellement la valeur du paramètre de la commande à partir du fichier de commande fourni à l'application par l'utilisateur final.

L'opérateur numérique de *Code_Aster* interroge le superviseur via l'API `getvxx` pour récupérer la valeur du paramètre à partir du nom du mot-clé. Le superviseur retourne la valeur fournie par l'utilisateur ou la valeur par défaut du paramètre.

Mot-clé facteur

Un mot-clé facteur est une chaîne de caractères identifiant un groupe de mot-clés simples sémantiquement associés. Un mot-clé facteur est défini à l'intérieur d'une commande. Une commande peut contenir plusieurs mots-clés facteurs éventuellement optionnels., chaque mot-clé facteur contenant lui-même des mots-clés simples de même nom.

L'utilisateur final pourra définir dans son fichier de commandes, un mot-clé facteur en spécifiant son nom puis sa **valeur** c'est-à-dire la valeur des paramètres numériques définis derrière les mots-clés simples du mot-clé facteur.

Le développeur de la fonctionnalité de *Code_Aster* définit les **caractéristiques** du mot-clé facteur dans le catalogue de la commande contenant le mot-clé facteur :

1) son nom (la chaîne de caractères utilisable par l'utilisateur final et par l'opérateur numérique),
2) ses règles de composition en mots-clés simples,
3) le statut du mot-clé facteur (facultatif ou obligatoire),
4) le nombre minimum de répétition du mot-clé facteur,
5) le nombre maximum de répétition du mot-clé facteur,
6) un commentaire explicatif en anglais et/ou en français,
7) la clé définissant le manuel et le chapitre consacré à ce mot-clé dans la documentation de *Code_Aster*.

Bloc conditionnel

Un bloc conditionnel (un bloc), associe :

1) des mots-clés simples,
2) des mots-clés facteurs,
3) et des blocs conditionnels.

L'occurrence du bloc dans sa commande, dépend d'une condition exprimée lors de la définition de la commande par le développeur de la fonctionnalité numérique.

Le développeur de l'opérateur correspondant à la commande contenant le bloc, spécifie les caractéristiques du bloc :

1) son nom,
2) sa condition,
3) ses règles de composition en mots-clés simples,
4) un commentaire explicatif en anglais et/ou en français.

L'utilisateur final pourra donner une valeur aux paramètres du traitement en utilisant les mots-clés facteurs et les mots-clés simples associés dans le bloc conditionnel mais sans spécifier le nom du bloc.

Règle de composition

Les entités composites du catalogue de commandes telles que « jeu de commandes », commande, mot-clé facteur et bloc conditionnel, structurent d'autres entités en suivant éventuellement une ou plusieurs règles de composition parmi les suivantes :

`AU_MOINS_UN`
: La règle `AU_MOINS_UN` exprime que l'une au moins des entités dont les noms sont passés en arguments doit être présente dans l'entité composite dans laquelle figure la règle.

`UN_PARMI`
: La règle `UN_PARMI` exprime que l'une et une seule des entités des entités dont les noms sont passés en arguments doit être présente dans l'entité composite dans laquelle figure la règle.

`EXCLUS`
: La règle `EXCLUS` exprime que si l'une des entités dont le nom est passé en argument, est présente, les entités correspondant aux autres arguments doivent être absentes dans l'entité composée dans laquelle figure la règle. Autrement-dit si plusieurs entités du groupe sont présentes la règle est violée.

`ENSEMBLE`
: La règle `ENSEMBLE` exprime que si l'une des entités dont le nom est passé en argument est présente dans l'entité composée, alors toutes celles correspondant aux autres noms devront l'être aussi. L'ordre des occurrences n'a pas d'importance. Et si aucune des entités représentées dans la règle n'est présente dans l'entité composite, l'entité composite est valide.

`PRESENT_PRESENT`
: La règle `PRESENT_PRESENT` exprime que si l'entité correspondant au **premier** nom est présente, alors toutes celles correspondant aux autres noms devront l'être aussi dans l'entité composite courante. L'ordre d'occurrence des autres entités n'a pas d'importance. Si aucune des entités représentées dans la règle n'est présente, l'entité composite est valide.

PRESENT_ABSENT

La règle PRESENT_ABSENT exprime que si l'entité correspondant au **premier** nom est présente, alors toutes celles correspondant aux autres noms devront être absentes dans l'entité composite courante. L'ordre d'occurrence des autres entités n'a pas d'importance. Si aucune des entités représentées dans la règle n'est présente, l'entité composite est valide.

Chaque règle de composition (appelée aussi simplement « règle ») est une classe (voir le module regle.py).

1.2 Prise en charge de la maintenance

La démarche suivante est proposée au candidat à la maintenance de l'interface graphique Eficas et/ou du superviseur de *Code_Aster*.

1) Etudier « Accas » c'est-à-dire ce qui est commun à l'interface graphique Eficas et au superviseur de *Code_Aster* ;
2) Etudier la structure du catalogue général des commandes : dans le fichier catalogue et dans la mémoire de l'application. Pour cela :

•se familiariser aux techniques de programmation en Python, utilisées dans Accas ;
•développer une petite maquette de factory (cf. [§4.1]) pour bien intégrer le mécanisme de base du chargement des mots-clés.

1) Etudier la structure du jeu de commandes (dans son fichier) et dans la mémoire ; en particulier la question du chargement du jeu de commande (mécanisme et zones de codes concernées) doit être considérée ;
2) Examiner les scripts Python ou les sources C mis en jeu à l'occasion de demandes de modification ou de traitement des erreurs détectées par les utilisateurs.

2 L'organisation sources Python

2.1 Conventions

Les conventions suivantes qui ont pour but de faciliter la lecture des scripts, sont imparfaitement appliquées

1) un nom de classe commence par une lettre majuscule ;
2) l'identificateur d'un objet de type Python list commence par le préfixe l_ (cette règle est utilisée dans Eficas) ;
3) dans les packages utilisés par le superviseur (Noyau, Execution, Validation, Build et Accas) une seule classe est définie par module c'est-à-dire par fichier *.py ;
4) dans les packages utilisés par le superviseur (Noyau, Execution, Validation, Build et Accas) le nom de chaque module commence par un préfixe indiquant le nom du package.
    1) N_ pour Noyau
    2) V_ pour Validation
    3) E_ pour Execution
    4) B_ pour Build
    5) A_ pour Accas

2.2 Typologie des modules Python

Chaque classe est définie dans un module : par exemple, la classe `MCSIMP` est définie dans le module `N_MCSIMP.py` où le préfixe `N_` désigne le nom du package (`Noyau`) contenant le module.

La technique des packages permet de découper les modules en fonction du domaine d'activité dans lequel il agit. A chaque domaine correspond un package Python

Par exemple, la classe MCSIMP existe dans chacun des cinq packages,

1) ·Noyau : `N_MCIMP.SIMP` ;
2) ·Validation : `V_MCSIMP.SIMP` ;
3) ·Ihm : `I_MCIMP.SIMP` ;
4) ·Accas : `A_MCIMP.SIMP` ;
5) ·Build : `B_MCIMP.SIMP` ;

`Noyau`

Ce package contient essentiellement le système de classe de la factory du jeu de commande.

`Validation`

Ce package contient les modules effectuant les vérifications de validité des objets (commandes, blocs conditionnels, mots-clés, …).

`Build`

Ce package n'est présent que dans le superviseur contient les modules traitant les commandes de type macro et les méthodes de requête au jeu de commande depuis l'API-C : les interfaces GETVxxx.

`Accas`

Ce package est le plus important. Il contient – en particulier - les classes les plus englobantes utilisées aussi bien par l'interface –graphique Eficas que par le superviseur de *Code_Aster*. C'est dans ce package qu'il faut rechercher les objets et les méthodes spécialisées dans le traitement – non graphique – des commandes :

1) chargement du catalogue ;
2) chargement d'un jeu de commande ;
3) exécution du jeu de commande.

Les classes filles définies dans ce package le sont par héritage de classes parentes ayant le même nom que les classes filles mais étant situées dans un package différent.

Ihm

Les classes de ce package enrichissent les classes du Noyau de méthodes – non liées à l'aspect graphique – utilisée par l'interface graphique Eficas.

Editeur

Ce package contient les modules de traitement graphique du jeu de commande.

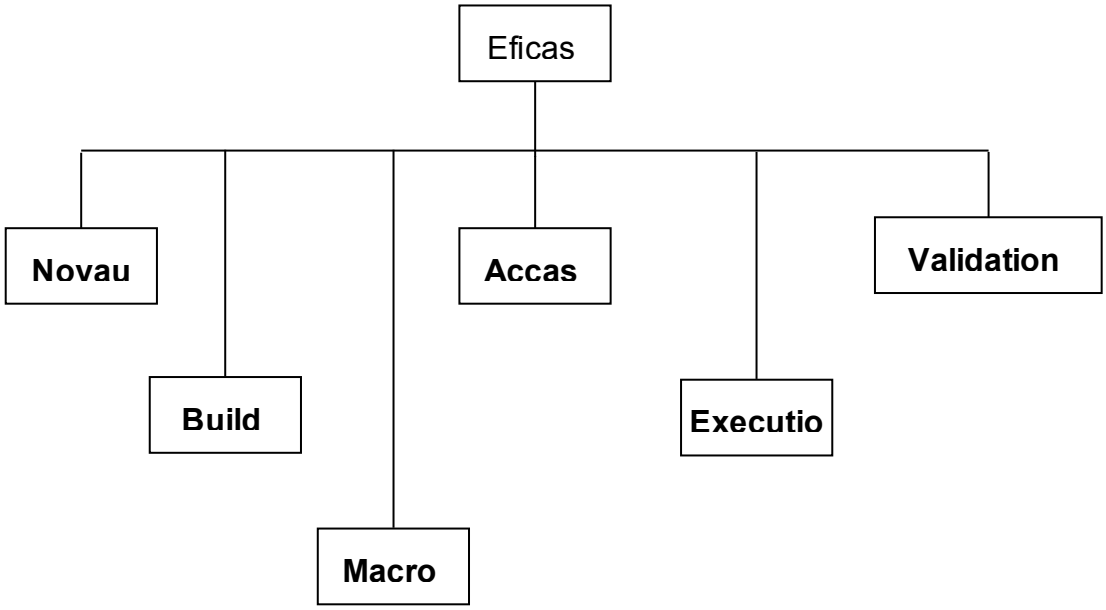
2.3 Hiérarchie des répertoires des sources

2.3.1 Les sources python du superviseur

Le superviseur de *Code_Aster* est composés de modules python écrits :

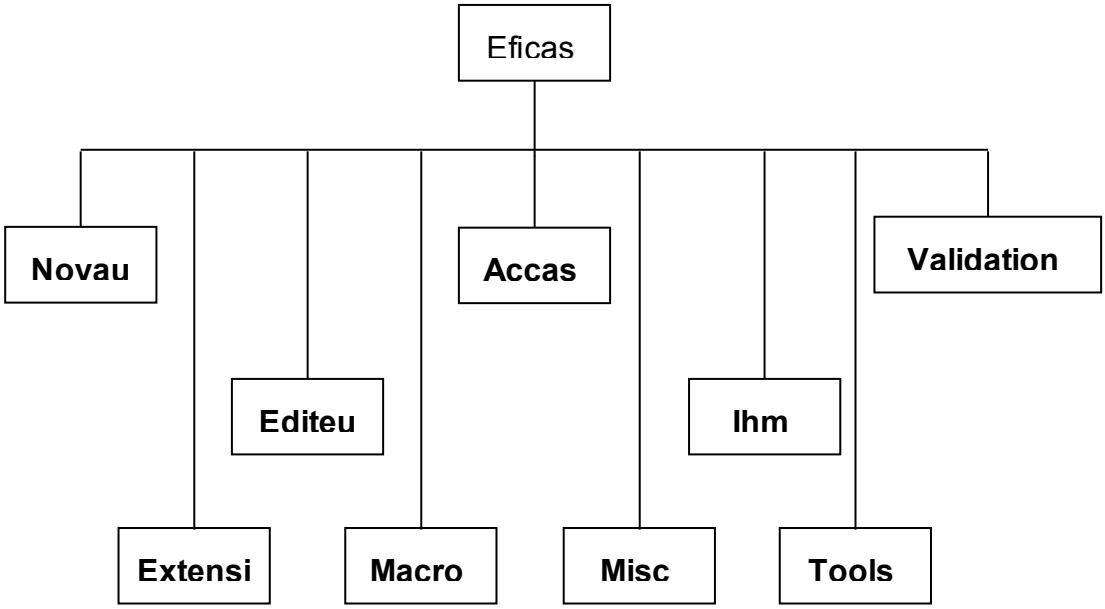
1) en C pour l'interface applicative du superviseur : le module `aster` (`astermodule.c`) ;
2) en Python pour la gestion du jeu de commande (chargement, organisation, …).

Le schéma ci-dessous présente les répertoires contenant les sources Python du superviseur.



2.3.2 Les sources de l'interface-graphique Eficas

Les sources sont organisés en répertoires sous le répertoire d'installation `Eficas` :



2.4 L'environnement

Ce paragraphe décrit les conditions pré-requises au fonctionnement du Superviseur et de l'interface graphique Eficas.

2.4.1 Installation de l'interpréteur Python

A renseigner

2.4.2 Paramètres shell de configuration

A renseigner

2.4.3 Installation de l'interface-graphique Eficas

A renseigner

2.4.4 Mise à jour du superviseur

A renseigner

3 Utilisation du langage Python dans Eficas et dans le superviseur de *Code_Aster*

On présente ici les techniques de programmation en langage Python, dont la maîtrise est nécessaire à tout candidat à la maintenance de l'interface graphique Eficas et du superviseur de *Code_Aster*.

3.1 Appel d'une fonction avec un nombre variable d'arguments

L'utilisateur final de *Code_Aster* utilise un script Python pour déclencher les fonctionnalités et fournir des valeurs aux paramètres de la fonctionnalité. La fourniture de ces valeurs étant parfois facultative, le noyau Accas utilise le mécanisme prévu dans le langage Python pour passer un nombre variable d'arguments à une fonction.

Pour maintenir Accas, il est, par conséquent, nécessaire de maîtriser cette technique dont nous présentons un petit exemple ci-dessous.

```
# script main.py
def fonc( nombre , *tup_args , **d_args ) :
    print nombre
    print repr(tup_args)
    print repr(d_args)

fonc(11111,"arg 2","arg 3",4,n=5,j=6)
```

Sous Unix, l'interprétation du script `main.py` se fait par :

```
$ python main.py
```

Elle donne le résultat suivant sur la sortie standard :

```
11111
('arg 2', 'arg 3', 4)
{'n': 5, 'j': 6}
```

Seul l'argument nombre est obligatoire. Les éventuels arguments positionnels suivant sont stockés dans un tuple (qui peut être vide) et les éventuels arguments passés par mot-clé sont stockés dans un dictionnaire.

Cette technique est utilisée en particulier, par les objets qui construisent le jeu de commande dans la mémoire puis qui l'initialisent.

3.2 Utilisation des espaces de nom

3.2.1 Notion d'espace de noms

Un *espace de noms* (voir [bib1], page 97) est un *dictionnaire Python* contenant un ensemble de couples nom/valeur. Le nom est en général une chaîne de caractères et la valeur peut être une valeur numérique, une fonction ou un objet.

Dans un module Python, chaque instruction est effectuée dans un espace de noms spécifique appelé **espace de noms local** dont le contenu peut être affiché par la fonction `locals()`. Les instructions ont également accès à l'**espace de noms global** dont le contenu peut être affiché avec la fonction `globals()`.

Accas utilise en particulier, un espace de noms pour stocker le dictionnaire des définitions qui sera utilisé pour interpréter les mots-clés et charger leur valeur dans la mémoire.

3.2.2 Le module __builtin__

Un module remarquable est le module standard __builtin__ qui est – sauf cas particulier ([bib2], page 100) – référencé dans chaque module utilisateur par l'attribut __builtins__ **en mode ReadOnly seulement**.

L'interprétation de la séquence suivante :

```
# script main.py
print globals()
```

affiche

```
{'__doc__':None,'__name__':'__main__','__builtins__':<module'__builtin__'(built-in)>}
```

Le module __builtin__ est importé par défaut ; toutes les données et les fonctions qu'il contient sont donc accessibles par défaut, dans tous les modules de l'application. Les données et les fonctions définies dans le module __builtin__ peuvent donc être considérées comme les plus globales à l'application.

Entre autres, on trouve dans ce module, les outils suivants :

1) les fonctions locals(). et globals() ;
2) la variable __debug__ qui conditionne l'interprétation suivant sa valeur 1 (défaut) ou 0 ;
3) la fonction « builtin » __import__.

```
# script main.py
# On installe le module context dans l'espace global de l'interpreteur
# (__builtin__)
# sous le nom CONTEXT afin d'avoir accès aux fonctions
# get_current_step, set_current_step et unset_current_step de n'importe où
import context
import __builtin__
__builtin__.CONTEXT=context
```

Il est possible d'importer et d'enrichir le module __builtin__ ; cette technique est utilisée dans Eficas dans le script Noyau/__init__.py.

3.2.3 Exécution d'une commande Python dans un espace de noms

Dans l'exemple suivant, la variable gb définie en début de script est accessible dans toutes les unités du module : elle est définie dans l'espace de noms global.

```
# 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'interprétation du script `main.py` affiche :

```
locals()={'b': 246, 'a': 123}
globals()={'__doc__':None, 'fonc': <function fonc at 0x810aee4>, 'z': -1 , 'x': 123,
'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'gb' : 2}
```

Un espace de noms spécifique peut être créé et utilisé pour exécuter des instructions stockées dans une chaîne de caractères.

```
# 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_contexte
```

Lorsqu'il interprète le script `main.py`, l'interpréteur enrichit le l'espace de noms `d_contexte` avec `'x':3` le résultat de l'instruction `s_commande`. Mais l'espace de noms local ne contient pas le résultat l'instruction `s_commande`.

Lors de l'utilisation de l'instruction `exec` , on peut spécifier que les données créées doivent l'être dans l'espace de noms local comme le montre l'exemple ci-dessous.

```
# script main.py
# script main.py
d_contexte={ 'a' : 1 , 'b' : 2 }
print d_contexte
s_commande='x=a+b'
exec s_commande in d_contexte, locals()
print 'x=',x
```

ce qui affiche :

```
{'b': 2, 'a': 1}
x= 3
```

L'instruction Python `exec` permet de créer un espace de noms par l'interprétation d'un script Python - stocké dans une chaîne de caractères - en spécifiant un espace de noms derrière le mot-clé `in`, par exemple : `exec s_script in espace`

3.3 Une deuxième manière d'importer un module

La fonction « builtin » `__import__` ([bib2], p. 100), permet d'importer un module à partir de son nom stocké dans une chaîne de caractères. En fait, cette fonction est appelée par l'interpréteur Python lors de l'import d'un module par l'instruction `import` ([bib3], p629). Cette technique est utilisée à plusieurs reprises dans Eficas (packages `Editeur` et `Extensions`).

```
nom_module='string'
print dir()
module_string =__import__( nom_module )
print dir()
print dir(module_string)
print module_string.lower( 'ABCD' )
```

On remarque que la fonction `__import__` retourne dans `module_string`, une référence sur le module `string` à partir de laquelle il est possible d'accéder aux outils contenus dans le module.

La fonction « builtin » __import__ peut aussi être utilisée pour importer un module spécifique à partir d'un paquetage (package). L'exemple suivant importe le catalogue général des commandes de *Code_Aster* puis affiche sur la sortie standard le nom de chaque commande.

```
import sys
TOP='/local/yessayan/Eficas/EficasPourSalome' # repertoire d'installation
sys.path.append( TOP )
sys.path.append( TOP+'/Aster' )
from Cata import cata
for commande in cata.JdC.commandes :
    print commande.nom
```

Il peut s'écrire :

```
import sys
TOP='/local/yessayan/Eficas/EficasPourSalome'
sys.path.append( TOP )
sys.path.append( TOP+'/Aster' )
package=__import__( 'Cata.cata' )
module_cata= getattr( package, 'cata' )
for commande in module_cata.JdC.commandes :
    print commande.nom
```

Ainsi qu'il est indiqué dans la documentation de python (http://www.python.org/doc/current/lib/built-in-funcs.html), l'instruction

```
__import__( 'Cata.cata' )
```

importe le package Cata.

Il reste donc à récupérer le module du catalogue à partir du package par exemple :

```
module=getattr(package, 'cata'))
```

La commande __import__ permet d'importer un module ou un package dont le nom n'est connu qu'au moment de l'interprétation du script courant.

3.4 Module Python écrit en langage C

A renseigner

4 Le catalogue général des commandes de *Code_Aster*

4.1 Exemple 1 : une factory pour construire les mots-clés simples

4.1.1 Principe de la factory

L'utilisateur final de *Code_Aster* fournit des valeurs aux fonctionnalités par l'intermédiaire de mots-clés. Un mot-clé permet d'introduire une valeur (mot-clé simple) ou un autre mot-clé (mot-clé facteur).

Le mécanisme de chargement et surtout d'utilisation de la valeur du mot-clé est compliqué par le fait que le type de cette valeur n'est pas unique. Ce type peut être par exemple : float, int, string, … Dans l'exemple suivant :

```
MASSE_VOLUMIQUE=7800.0
FICHIER_MAILLAGE='maillage.unv'
```

Les mot-clés simples MASSE_VOLUMIQUE et FICHIER_MAILLAGE reçoivent respectivement la valeur 7800.0 et 'maillage.unv'

Il faut donc décrire quelque part les caractéristiques du mot-clé simple (nom, type, valeur par défaut, unité, statut facultatif ou obligatoire, …). La réponse à cette question est fondée sur la séparation caractéristiques/valeur suivante :

1) le rôle principal d'un objet de type MCSIMP est d'envelopper la **valeur d'un mot-clé simple** autrement-dit l'attribut principal d'un objet de type MCSIMP est une valeur ;
2) un objet de type SIMP a deux rôles principaux :

•envelopper la définition d'un mot-clé simple : un objet SIMP contient l'ensemble des caractéristiques d'un mot-clé simple : son nom, le type de sa valeur, … ;
•mais cet objet dispose aussi d'une fonction __call__ qui permet de générer un mot-clé simple à partir de ses caractéristiques ; un objet SIMP peut être considéré comme une machine à fabriquer un objet de MCSIMP d'où le terme *factory*

Chaque objet MCSIMP contient sa valeur et une référence sur l'objet SIMP qui décrit ses caractéristiques.

La technique de la factory est présentée ci dessous en réduisant les classes MCSIMP et SIMP à leur minimum.

1) La classe MCSIMP a deux attributs :

•definition : sa définition (une référence sur l'objet SIMP qui a créé l'objet MCSIMP) à partir de laquelle il est possible de récupérer la partie texte du mot-clé : definition.nom ou son type definition.type ;
•val : sa valeur

1) la classe SIMP a deux attributs :
   1) nom : la partie texte (chaîne de caractère pouvant contenir un espace blanc) du mot-clé MCSIMP que l'objet SIMP va construire ;
   2) type : le type de la valeur introduit par le mot-clé MCSIMP que l'objet SIMP va construire.

La fonction __call__ de la classe SIMP crée un objet de type MCSIMP. Cela implique que la classe MCSIMP soit définie avant la définition de cette fonction.

Dans le modèle de conception factory appliqué au jeu de commandes de *Code_Aster*, la classe fabriquée (exemple MCSIMP) doit être définie avant la classe fabricante (SIMP).

```
# script main.py
import sys
# string en python 2.1 devient str à partir de python 2.2
s_version=str(sys.version_info[0])+'.'+str(sys.version_info[1])
assert(float(s_version)<=2.1),'la  version  '+s_version+'  de  python  est
INVALIDE'

class MCSIMP :
    def __init__( self, definition ,val ,parent=None ) :
        self.definition = definition
        self.val        = val
        self.parent     = None

class SIMP :
    def __init__( self, nom, type ) :
        self.nom = nom
        self.type =type
    def __call__ ( self , val , parent=None ) :
        assert( str(type(val))=="<type '"+self.type+"'>" )
        return MCSIMP( definition=self, val=val , parent=parent)

d_context={
        'MASSE_VOLUMIQUE'  : SIMP(nom='MASSE VOLUMIQUE',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 est ajouté à l'espace de noms
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  est  ajouté  à  l'espace  de
noms locals()
print mail
print mail.definition.nom
print mail.val

sys.stderr.write( "FIN NORMALE de main.py"+'\n' )
```

L'interprétation du script `main.py` ci-dessus donne l'affichage suivant :

```
<__main__.MCSIMP instance at 0x810c4ac>
MASSE VOLUMIQUE
7800.0
<__main__.MCSIMP instance at 0x810c4d4>
FICHIER MAILLAGE
maillage.d
```

Considérons les lignes

```
s_commande = 'rho=MASSE_VOLUMIQUE( 7800.0 )'
exec s_commande in d_context, locals()
```

Le mécanisme de construction de l'objet rho, dans l'espace de noms local, est un mécanisme de factory comprenant les étapes suivantes :

1) Dans l'espace de noms d_context, la commande devient :
rho=SIMP(nom='MASSE VOLUMIQUE',type='float')(7800.0)
puis elle est exécutée ;
1) Un objet de type SIMP est construit avec le nom 'MASSE VOLUMIQUE' ;
2) la méthode __call__ est appelée avec val=7800.0 en argument ;
3) à partir du nom (self.nom) et de la valeur (val), la méthode __call__ crée et retourne un objet de type MCSIMP ;
4) l'objet retourné est affecté à la variable rho dans l'espace de noms local.

Important :

Dans la commande s_commande , les mot-clés ne doivent pas contenir d'espace blanc :
" rho=MASSE_VOLUMIQUE(7800.0) "est une instruction Python valide.

Un espace blanc dans "rho=MASSE VOLUMIQUE(7800.0)" engendrerait une erreur à l'interprétation.

Dans le dictionnaire des mots-clés les chaînes de caractères utilisées pour les clés, doivent obéir aux règles d'écriture d'un identificateur Python : pas d'espace blanc.

4.1.2 L'organisation de la factory en fichiers

L'organisation en fichiers, présentée ci-dessous, décrit celle qui est utilisée pour l'interface graphique Eficas et pour le superviseur. Elle décrit également la procédure de chargement du jeu de commande dans la mémoire à partir des informations fournies – dans le fichier de commande - par l'utilisateur final.

Les fichiers – tous des scripts Python - doivent être définis dans l'ordre suivant :

- MCSIMP. py
- SIMP.py
- dictio.py : le dictionnaire des mots-clés MASSE_VOLUMIQUE et FICHIER_MAILLAGE
- valeurs.py : le fichier des valeurs fournies par l'utilisateur final (pour Aster, le fichier de commandes utilisateur)
- main.py : le code chargeant les valeurs fournies par l'utilisateur final en lisant et interprétant le fichier valeurs.py (pour Aster l'éxecutable, aussi interpréteur python : aster.exe)

Rappelons que la définition de la classe MCSIMP doit être effectuée avant celle de la classe SIMP. Ce qui conduit à une organisation commençant par la définition de la classe MCSIMP.

```
# script MCSIMP.py

class MCSIMP :
    def __init__ ( self , definition , val , parent=None ) :

        assert(definition.__class__.__name__=='SIMP')
        self.definition = definition

        assert(type(val).__name__==definition.type)
        self.val = val
        self.parent =None

        return
```

L'attribut `self.parent` inutile ici, sera utilisé lorsque le mot-clé sera défini à l'intérieur d'une commande. Il contiendra alors une référence sur cette commande.

Le développeur d'Accas peut ensuite définir la classe `SIMP`.

```
# script SIMP.py

import MCSIMP
import types

class SIMP :
    def __init__( self , nom , type ) :
        self.nom = nom
        self.type =type
    def __call__ ( self , val , parent=None ) :
        assert( str(type(val))=="<type '"+self.type+"'>" )
        return MCSIMP.MCSIMP( definition=self, val=val , parent=parent)
```

Une fois les deux modules `SIMP` et MCSIMP mis à sa disposition, le développeur de fonctionnalités numériques du code peut maintenant définir un "catalogue de mots-clés applicatif ' dans le fichier `dictio.py`. Ce script définit dans un dictionnaire python, la description (type, valeurs possibles, domaine de définition, …) des valeurs associées au mot-clés de sorte à permettre la lecture de ces valeurs.

Par exemple :

```
# script dictio.py

from SIMP import SIMP
dict={'MASSE_VOLUMIQUE':SIMP(nom='MASSE VOLUMIQUE',type='float',
           'FICHIER_MAILLAGE':SIMP(nom='FICHIER MAILLAGE',type='string')}
```

Et l'utilisateur final peut enfin utiliser les fonctionnalités en fournissant un script, par exemple `valeurs.py`.

```
# script valeurs.py

rho=MASSE_VOLUMIQUE( 7800.0 )
mail=FICHIER_MAILLAGE( 'maillage.d' )
```

Pour charger en mémoire les données fournies par l'utilisateur, le code lira le fichier de commandes comme suit :

```
# script main.py

from SIMP import *


d_context={'MASSE_VOLUMIQUE':SIMP(nom='MASSE VOLUMIQUE',type='float') ,
           'FICHIER_MAILLAGE':SIMP(nom='FICHIER MAILLAGE',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
```

Pour charger en mémoire, la valeur associée à un mot-clé simple, il faut interpréter le script python – texte du mot-clé simple et valeur(s) associée(s) fournis par l'utilisateur final - dans l'espace de noms (le dictionnaire `d_context`) du mot-clé.

4.2 Exemple 2 : une factory pour construire une commande

En pratique, les mots-clés ne sont pas définis isolément mais obligatoirement à l'intérieur d'une commande. Ce qui complique le processus de construction des mots-clés dans la mémoire. Nous présentons maintenant un exemple – toujours simplifié – destiné à faciliter la compréhension de ce processus.

Pour cela nous allons considérer qu'un jeu de commande est une liste de commandes de type « procédure » et que chaque commande est paramétrée par des mot-clés simples et uniquement par des mots-clés simples (pas de bloc conditionnel, pas de mit-clé facteur). Cette simplification accroît la lisibilité tout en conservant toutes les catégories de difficultés à affronter pour charger le jeu de commande en mémoire.

On commence donc par ajouter une factory de commandes de type `PROC_ETAPE`.

4.2.1 Ajout des classes PROC et PROC_ETAPE à la factory

La classe PROC_ETAPE qui modélise une commande de type « procédure », est très proche de la classe MCSIMP. A tel point que toutes deux pourraient hériter d'une classe mère commune (c'est d'ailleurs le cas dans Accas).

```
# script PROC_ETAPE.py

print "\tImport de "+__name__

class PROC_ETAPE :
    def __init__ ( self , definition , args={} ) :
        print 2*'\t'+"PROC_ETAPE __init__ : creation d'un objet "+\
        self.__class__.__name__
        print 3*'\t'+'PROC_ETAPE __init__ : definition.nom=',definition.nom
        print 3*'\t'+'PROC_ETAPE __init__ : args=',args

        assert(definition.__class__.__name__=='PROC')
        self.definition = definition
        self.valeur = args

        return
```

Un objet PROC_ETAPE est fabriqué par un objet de type PROC. Son attribut self.definition est une référence sur l'objet PROC qui l'a créé.

```
# script PROC.py
print "\tImport de "+__name__
import PROC_ETAPE
from SIMP import *

class PROC :
    def __init__ ( self, nom, op , **args ) :
        print 1*'\t'+"PROC __init__ : creation d'un objet "+\
        self.__class__.__name__
        print 2*'\t'+'PROC __init__ : nom=',nom
        print 2*'\t'+'PROC __init__ : args=',args
        self.nom = nom          # texte de la commande
        self.entites = args     # dictionnaire des fabricants
        self.op = op            # numero de l'operateur fortran
        return

    def __call__ ( self , **args ) :
                #  args  contient  la  definition  des  valeurs  des  MCSIMP
(MASSE_VOLUMIQUE,
        # FICHIER_MAILLAGE)
        print 1*'\t'+'PROC __call__ : args=',args
        print 1*'\t'+'PROC __call__ : self.entites=',self.entites
        # construction des mots-clés simples de la commande et ajout dans
        # le dictionnaire des mots-cles de la commande PROC en cours de
        # 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( self, dict )
```

Lors de sa création effectuée à partir du catalogue, l'objet PROC mémorise dans le dictionnaire self.entites, la composition de la commande en mots-clés simples ; ces information seront utilisées dans un deuxième temps – pour construire les mots-clés simples de la commande – lorsque l'objet PROC sera invoqué via sa méthode __call__.

C'est aussi la méthode __call__ qui initialisera les mot-clés situés à l'intérieur de la commande avec les valeurs fournies dans le fichier de commande (le module SIMP est exactement celui présenté dans le premier exemple).

4.2.2 Notion de catalogue de commandes

Dans le premier exemple ([§4.1]), la description des mots-clés était faite dans le dictionnaire(l'espace de noms) d_context. Mais pour faciliter la tâche des développeurs de *Code_Aster*, il est préférable de décrire les commandes et leur contenu par l'intermédiaire d'un script Python puis de convertir ce script en un dictionnaire qui servira d'espace de noms pour le chargement des commandes.

Pour notre second exemple, le catalogue peut s'écrire ainsi ;

```
# script cata.py

print 1*'\t'+"Import de "+__name__

from SIMP import *
from PROC import *

AFFE_MATERIAU=PROC( nom='AFFE_MATERIAU',
                    op=10,
                    MASSE_VOLUMIQUE=SIMP(nom="MASSE VOLUMIQUE",type='float'),
                    FICHIER_MAILLAGE=SIMP(nom="FICHIER MAILLAGE",type='string') )
```

Ce catalogue ne contient qu'une commande : AFFE_MATERIAU dont l'usage déclenchera l'appel à la routine Fortran op0010. Cette routine utilisera deux paramètres MASSE_VOLUMIQUE et FICHIER_MAILLAGE

La conversion du catalogue en un dictionnaire se fait simplement en important le catalogue dans un espace de noms. Ce que fait la séquence suivante :

```
d_context={}
string_cata="from cata import *"
exec string_cata in d_context
```

4.2.3 Notion de fichier de commandes

Le fichier de commandes est lui aussi un script Python, très simple destiné à être interprété dans l'espace de noms du catalogue de commandes. Par exemple :

```
# script commandes.py

AFFE_MATERIAU( MASSE_VOLUMIQUE=7800.0, FICHIER_MAILLAGE="maillage.unv" )
```

Dans le script commandes.py ci-dessus, l'utilisateur final demande l'exécution de la routine Fortran op0010 avec MASSE_VOLUMIQUE=7800.0 et FICHIER_MAILLAGE="maillage.unv". Il est interprété par la séquence suivante :

```
f_commandes=open( 'commandes.py' , 'r' )
string_commandes = f_commandes.read()
f_commandes.close()
exec string_commandes in d_context
```

A la fin de laquelle le jeu de commande (ici réduit à la seule commande AFFE_MATERIAU) est défini dans l'espace de noms d_context.

4.2.4 Utilisation du catalogue et du fichier de commande

Le script Python suivant effectue le chargement du catalogue, le chargement du fichier de commande et l'examen du jeu de commande dans la mémoire.

```
# script main.py
import traceback
trace=traceback.extract_stack()
script_file=trace[0][0]
prefixe=script_file+' : '
print prefixe+"DEBUT de ",script_file

d_context={}
# 1. Chargement du catalogue
# Creation - en important le catalogue cata - d'un espace de nom servant
# pour l'interpretation du jeu de commande

print 3*'\n'+prefixe+"import du catalogue"
string_cata="from cata import *"
exec string_cata in d_context

# 2. Chargement du texte des commandes
# Lecture du fichier de commmandes (le texte des commandes est stocke dans
une
# chaine de caratères)

print 3*'\n'+prefixe+"lecture du texte des commandes"
f_commandes=open( 'commandes.py' , 'r' )
string_commandes = f_commandes.read()
f_commandes.close()

# 3. Creation du jeu de commandes
# Interpretation du texte des commande dans le d_contexte du catalogue. La
structure
# jeu de commande, produite, est stockée dans l'espace de nom d_context
print 3*'\n'+prefixe+\
"Conversion du texte des commandes (string) en un jeu de commandes
(d_context)"

exec string_commandes in d_context

# 4. Parcours de la structure jeu de commandes dans le d_contexte

print 3*'\n'+prefixe+"Affichage du jeu de commandes"
import types
for k,v in d_context.items() :
                if type(v)==types.InstanceType and v.__class__.__name__ ==
```

```
'PROC_ETAPE' :
                # si l'attribut est une commande, on examine sa valeur
                # c'est-a-dire ses mots-clés
                print 1*'\t'+v.definition.nom+'\t'+str(v.__class__)
                for kk,vv in v.valeur.items() :
                        print 2*'\t'+kk,' : ',vv,'\t',vv.val

print 2*'\n'
print prefixe+"FIN NORMALE de",script_file
```

4.3 Le fichier catalogue

4.3.1 Catalogue d'une commande

Le catalogue d'une commande contient la description de la commande. Chaque commande est un instance de la classe OPER, PROC ou MACRO

Attribut	Description
nom	nom de la commande (chaîne de caractères sans espace blanc)
op	numéro de l'opérateur Fortran : entier compris entre 1 et 199
sd_prod	type du résultat, pour les commandes de type OPER
regle	liste des règles de composition de la commande
fr	commentaire en français
doc	référence de la documentation Aster
reentrant	
repetable	
entites	Composition de la commande : arguments contenant la description des mots-clés utilisés pour fournir des valeurs aux mot-clés de la commande

Exemple

```
ASSE_MAILLAGE=OPER(nom='ASSE_MAILLAGE',op= 105,sd_prod=maillage,
                   fr='Assembler deux maillages sous un seul nom',
                   docu='U4.23.03-e',reentrant='n',
         MAILLAGE        =SIMP(statut='o',typ=maillage,min=2,max=2 ),
) ;
```

Dans cet exemple, tiré du vrai catalogue de commandes de *Code_Aster* :

1) la commande décrite s'appelle ASSE_MAILLAGE ;
2) elle permet de déclencher l'opérateur Fortran op0105 ;
3) elle retourne une donnée de type maillage ; ce type est défini au début de catalogue cata.py ;
4) l'utilisation de l'opérateur Fortran op0105 requiert obligatoirement, la fourniture de 2 données de type maillage

4.3.2 Structure générale du fichier catalogue

Le fichier catalogue des commandes de *Code_Aster* – le module `$TOP/Aster/Cata/cata.py` - contient les informations suivantes :

1) import de toutes les informations du module `Accas`, en particulier `Accas.A_ASSD.ASSD`
2) déclaration de types dérivant du type générique `Accas.A_ASSD.ASSD`, utilisés pour typer les valeurs des mots-clés ou les valeurs retournées par les commandes ; par exemple les types :

•`entier`, `reel`, `complexe`, `liste`, `chaine` ;
•les géométriques, `no` (nœud), `groupno`, `ma` (maille), `groupma` ;
•`maillage`, `modele`, `mater`
•*etc*

1) la liste des catalogues des commandes c'est-à-dire la description de toutes les commandes, avec pour chaque commande

4.4 Installation du catalogue dans la mémoire

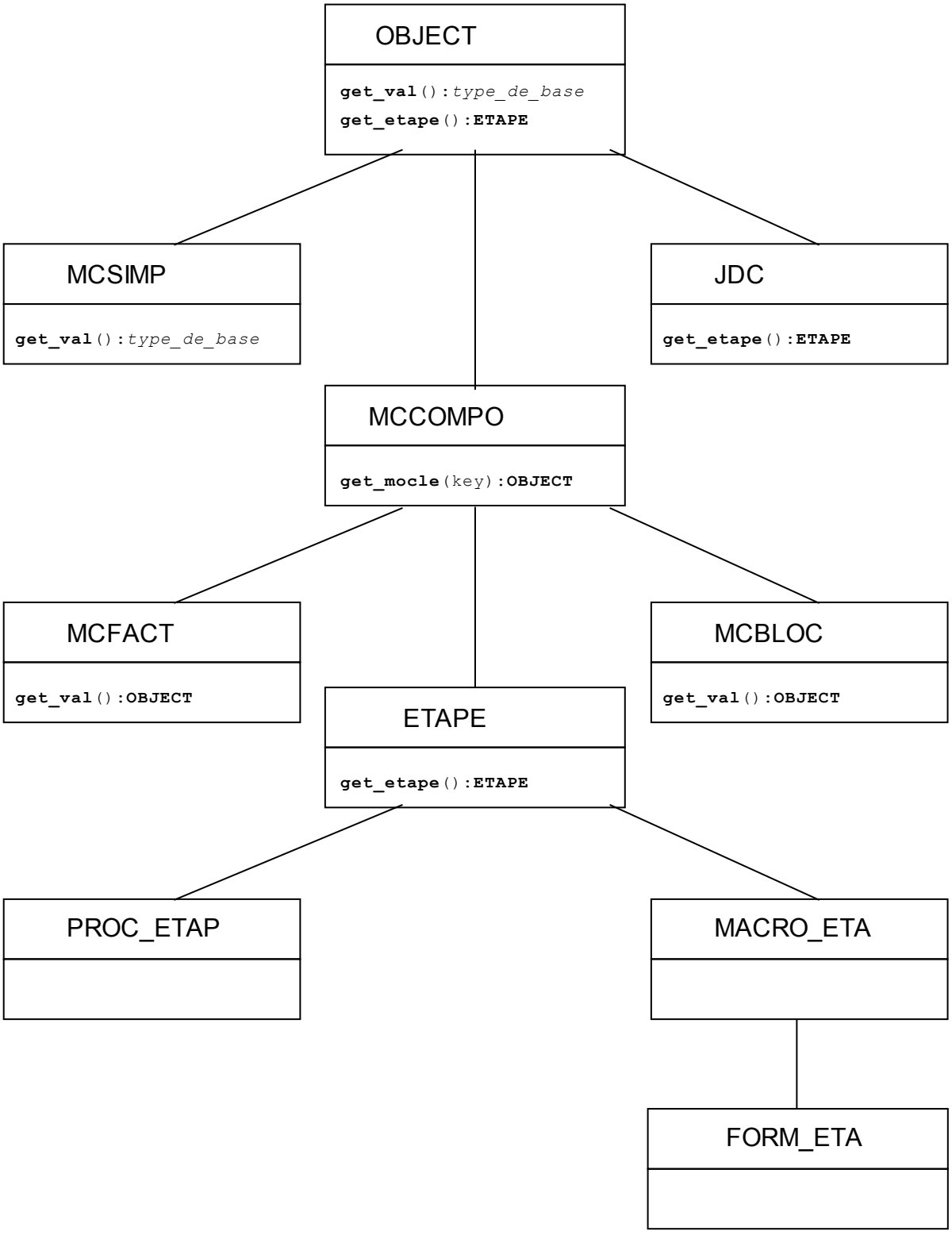
Il est important de se souvenir qu'une référence sur le catalogue courant est stockée dans le module dont la référence est stockée dans `CONTEXT`. La référence `CONTEXT` est elle-même définie dans l'espace de noms global `__builtin__`

Une référence sur le catalogue courant est obtenue par :

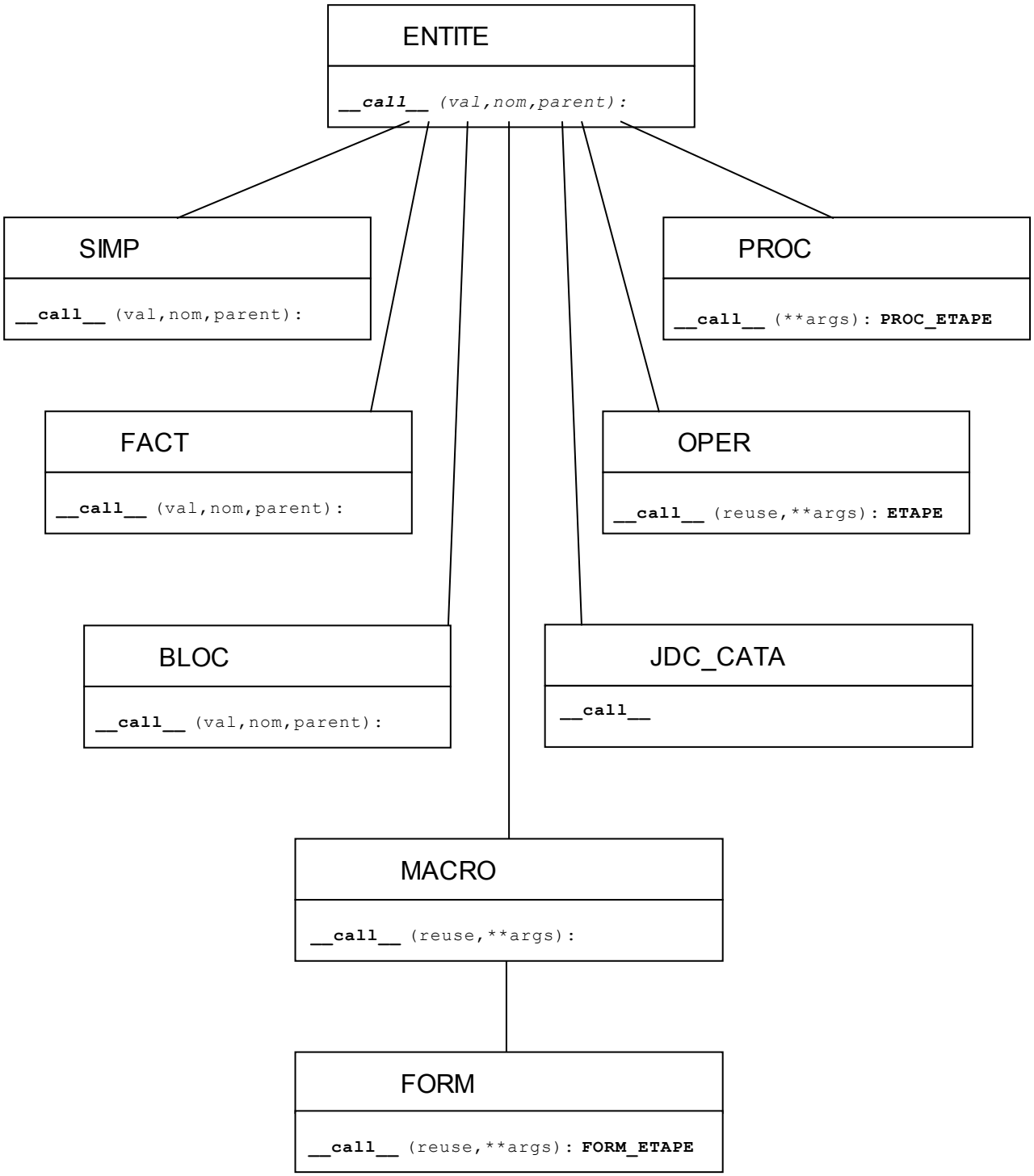
```
CONTEXT.get_current_cata()
```

4.4.1 La structure de données dans le package Noyau

Les classes suivantes ont pour rôle de stocker le jeu de commande dans la mémoire et de restituer la valeur des mots-clés à la demande.



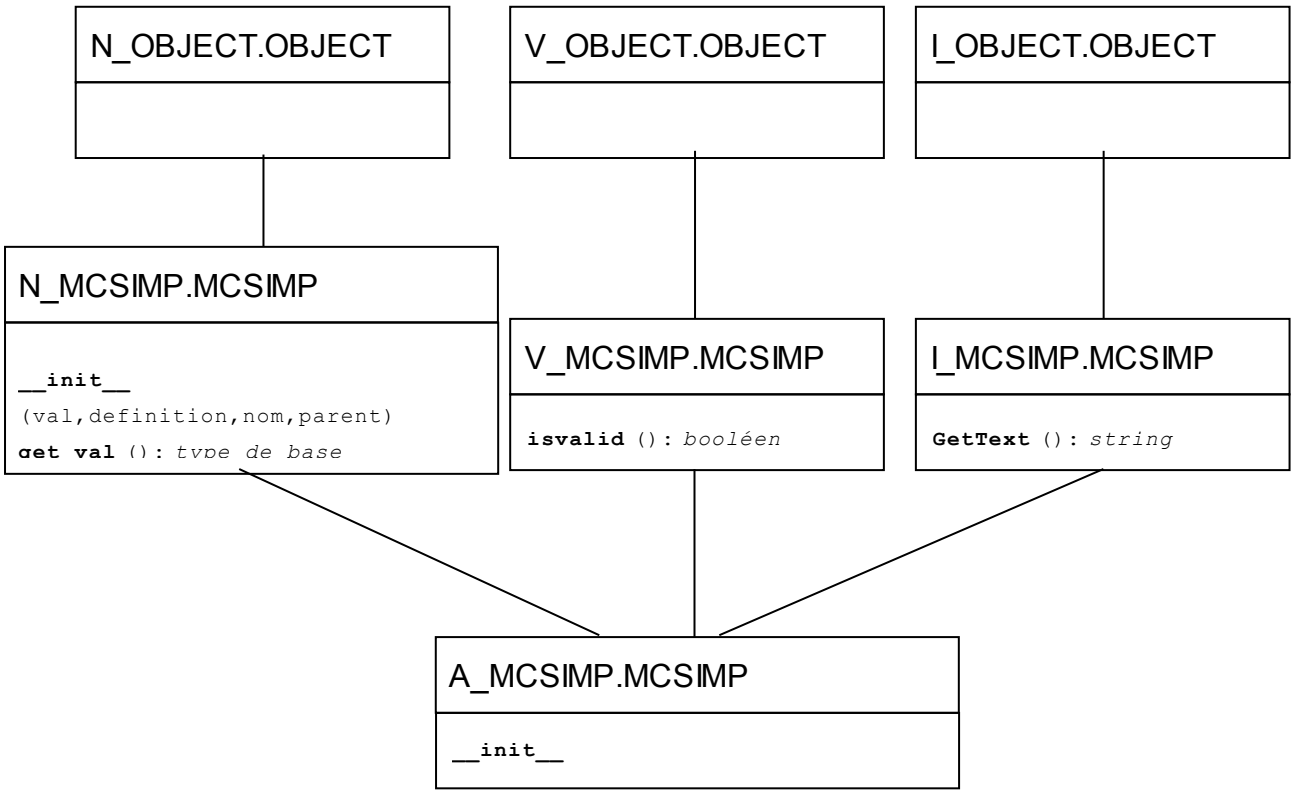
L'ensemble des classes présentées ci-dessous participent au chargement du jeu de commande dans la mémoire. Elles héritent toutes de la classe abstraite ENTITE. Et leur fonction principale est la méthode __call__ qui assure le rôle de fabrique (factory) d'objet.



4.4.2 Le package Accas

Le package Accas est le package principal utilisé par le superviseur de *Code_Aster* et par l'interface graphique Eficas. Il contient les modules correspondant aux classes obtenues par assemblage (utilisation de l'héritage multiple) des classes des autres packages.

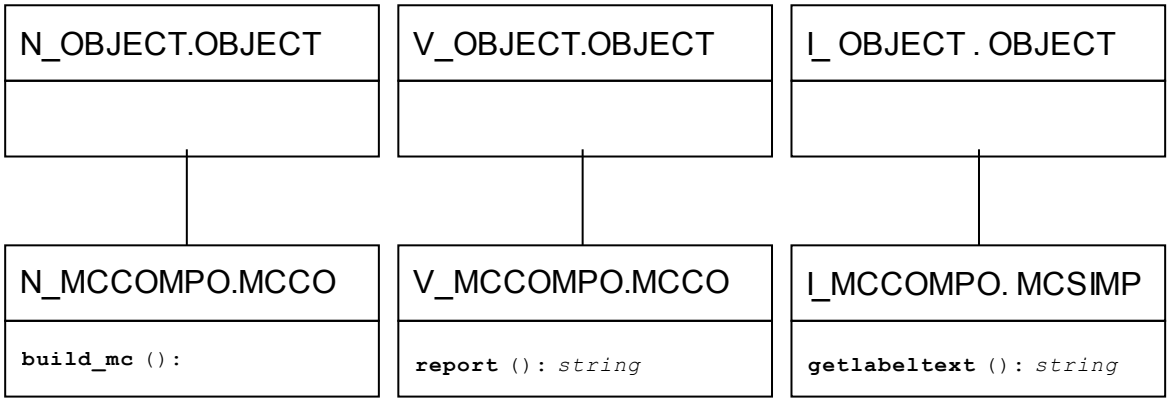
La classe MCSIMP



Les objets de la classe MCSIMP sont construits avec le constructeur de la classe A_MCSIMP.MCSIMP. Le schéma ci-dessus représente la principale contribution de chaque package aux fonctionnalités de la class MCSIMP
La méthode get_val() est utilisée par le superviseur pour récupérer la valeur du mot-clé simple et la retourner à l'opérateur de *Code_Aster* qui la demande.

La classe MCCOMPO

La classe MCCOMPO – qui modélise un OBJECT composite - n'existe pas dans le package Accas mais elle est importante car elle sert de base aux classes MCFACT, MCBLOC.



Le schéma ci-dessus met en évidence des comportements justifiant l'existence de la classe MCCOMPO.

1) la méthode build_mc() dans le package Noyau : elle construit les objets situés à l'intérieur de l'OBJECT ;
2) la méthode report() qui retourne le rapport de validation en appliquant la méthode isvalid() à tous les OBJECT situés dans le MCCOMPO.

5 Les macro-commandes Python

Description sommaire de la nature des macro-commandes :

1) Une macro-commande Python peut produire un ou plusieurs résultats (appelés concepts) alors que les commandes simples produisent zéro (commande de type PROC) ou un résultat (commande de type OPER) ;
2) Une macro-commande a des paramètres comme une commande ordinaire ; ce sont des mots-clés facteurs et des mots-clés simples ;
3) Le principal concept produit par une macro est retourné par la macro tandis que les concepts produits secondaires sont des arguments modifiés par la macro ;
4) Les concepts produits secondaires doivent être typés : cela est fait par l'intermédiaire d'une fonction fournie par le développeur de la macro par l'intermédiaire de l'argument sd_prod de la macro ;
5) Le corps de la macro-commande est une fonction Python prenant en charge le traitement qui inclut l'appel à d'autres commandes (ou même à d'autres macro-commandes).

Pour définir une macro-commande, son développeur doit donc définir :

1) les mots-clés de la commande ;
2) le type des concepts produits ;
3) le corps de la macro-commande.

6 Quelques questions

6.1 Comment savoir quel fichier catalogue est utilisé par un calcul ?

La situation est la suivante :

1) un calcul a été exécuté avec *Code_Aster* ;
2) plusieurs fichiers de catalogue des commandes sont présents dans l'environnement ;
3) l'utilisateur – ou le développeur – veut savoir celui qui est effectivement utilisé.

Une solution peut être la suivante :

1) importer le catalogue dans le fichier de commande, par exemple dans `ahlv100a.comm` ;
2) insérer dans le fichier de commande la séquence suivante qui :

•importe le catalogue,
•écrit la référence du catalogue sur la sortie standard,
•interrompt le traitement.

```
import cata, sys
print 'ahlv100a.comm:cata= ',cata
sys.exit(0)
```

Avec cette séquence, on obtient un résultat du style suivant :

```
cata=<module 'cata' from '/home/salome/yessayan/Devel/Asterv7/bibpyt/Cata/cata.pyc'>
```

D'où l'on déduit que le catalogue utilisé peut se trouver dans le fichier :

```
/home/salome/yessayan/Devel/Asterv7/bibpyt/Cata/cata.py
```

6.2 Comment le mode DEBUG est-il géré ?

Dans Eficas et dans le superviseur - en fait, dans tout script Python - le mode DEBUG est géré via une variable standard définie dans l'espace de noms global `__builtin__` : `__debug__`. En mode normal d'interprétation (`python main.py`), `__debug__` est mise à **1** (dans `main.pyc`) mais en mode optimisé (`python -o main.py`) `__debug__` est mise à **0** (dans `main.pyo`)

A tout instant, dans tout les modules, la variable __debug__ peut être utilisée pour conditionner le traitement.

6.3 Où le catalogue de commande est-il chargé dans la mémoire

Que ce soit dans le superviseur ou dans l'interface graphique Eficas, `JdC`, l'objet Python contenant le catalogue est créé dans module `cata` du package `Cata`. Plus exactement, `JdC` est créé au moment où le module `cata` est importé : l'import est réalisé dans la méthode `imports` de la classe `SUPERV` module `Execution/E_SUPERV`. Après l'import, l'objet `JdC` contient – dans son attribut `commandes`, de type `list` - la définition de toutes les commandes disponibles ainsi que celle de tous les mots-clés associés à chaque commande.

6.3.1 Création de l'objet JdC

Au début du script cata.py , le `JdC` est déclaré par l'instruction :

```
JdC = JDC_CATA(code='ASTER',
               execmodul=None,
               regles = (AU_MOINS_UN('DEBUT','POURSUITE'),
                         AU_MOINS_UN('FIN'),
                         A_CLASSER(('DEBUT','POURSUITE'),'FIN')))
```

Cette instruction fait principalement appel à la méthode `__init__` de la classe `N_JDC_CATA.JDC_CATA` (package `Noyau`). Dans cette méthode, l'objet `JdC` créé est enregistré dans la l'espace global `__builtins__` , par l'intermédiaire de la variable _cata dans le module `CONTEXT` : `__builtins__[ "CONTEXT" ]._cata`.

Une référence sur le catalogue courant est toujours disponible dans l'espace de noms global `__builtins__`.

6.3.2 Chargement des entités du catalogue dans l'objet JdC

Après la création le chargement s'effectue toujours au moment de l'import du catalogue dans la méthode `imports`, en créant des objets des types

1) OPER :
2) PROC :
3) MACRO :

6.4 Où le jeu de commandes est-il exécuté par le superviseur ?

Le jeu de commande `j` (objet de type `Accas.A_JDC.JDC`) est exécuté dans la méthode `Execute` de la classe `SUPERV` dans le module `E_SUPERV` du package `Execution`.

Deux cas sont possibles :

1) En mode PAR_LOT='OUI' (dans le script l'attribut j.par_lot du jeu de commande est positionné à 'OUI'), le traitement est effectué par l'appel

```
j.exec_compile() ;
```

1) En mode PAR_LOT='NON' (dans le script l'attribut j.par_lot du jeu de commande est positionné à 'NON'), le traitement est effectué par l'appel

```
ier= self.ParLotMixte( j ).
```

6.5 A quoi sert le mot-clé _F utilisé dans le fichier de commande ?

Dans le fichier de commande, un mot-clé facteur est introduit par la chaîne de caractères _F. En fait cette chaîne de caractères est un nom de classe qui prend en charge la création en mémoire du dictionnaire correspondant au mot-clé facteur à partir d'une description utilisant le signe égal '=' et des parenthèses plutôt que les deux points ':' et les accolades qu'il faudrait utiliser avec un dictionnaire Python standard.

Par exemple :
```
ELAS= _F(  E = 2.1E11,  NU = 0.3, ALPHA = 1.E-5,  RHO = 8000. )
```
est équivalent à :
```
ELAS={ E : 2.1E11,  NU : 0.3, ALPHA : 1.E-5,  RHO : 8000.}
```

Cette présentation est davantage adaptée aux souhaits des utilisateurs finals et à la tradition du langage de commande de *Code_Aster*.

6.6 Où l'interface getvxx du jeu de commande se trouve-telle ?

Les méthodes getvxx appartiennent à la classe ETAPE définie dans le module B_ETAPE du package Build.

7 Bibliographie

[1] Introduction à Python, Mark Lutz & David Ascher, O'REILLY, Paris, 2001
[2] Python, Essential Reference, David M. Beazley, New Riders, 2001

[3] Python 2.1 Bible, Dave Brueck & Stephen Tanner, 2001