Méthodes Python d'accès aux objets *Aster*

Résumé :

Ce document présente les méthodes Python permettant d'accéder aux informations contenues dans les structures de données Aster. Cette information peut être traitée par une programmation python, ou servir pour l'enchaînement conditionnel des commandes suivantes.

Table des matières

1 Introduction et précautions d'emploi....3
2 Les tables....4
2.1 Impression....5
2.2 Création ou impression d'une sous-table extraite par filtre....5
2.3 Tris....6
2.4 Accès aux valeurs....6
3 Méthodes d'accès sur les concepts....7
3.1 Accès aux contenu d'une SD liste....7
3.2 Accès au contenu d'une SD fonction ou d'une SD nappe....7
3.3 Évaluation d'une SD fonction ou formule....7
3.4 Accès au contenu d'une SD maillage....7
3.5 Accès au contenu d'une SD matr_asse_*....8
3.6 Accès aux clés d'une SD resultat....8
3.7 Accès au contenu d' une SD tran_gene....9
3.8 Accès au contenu d' une SD melasflu....9
3.9 Récupération en python des champs par éléments et champs aux nœuds (EXTR_COMP)....10
3.9.1 Arguments de la commande EXTR_COMP....10
3.9.2 Exemple d'utilisation....10
3.9.3 Résultats de la commande EXTR_COMP....11
4 Méthode d'accès à une structure de données quelconque....11
4.1 Accès à une structure de données de type vecteur....11
4.2 Accès à une structure de données de type collection....12

1 Introduction et précautions d'emploi

Dans *Code_Aster*, la plupart des commandes sont programmées en fortran. Les structures de données produites ne sont accessibles que par l'intermédiaire du gestionnaire de mémoire JEVEUX, lui même écrit en fortran. Dans une exécution standard du code, seuls les noms des concepts (et non des objets portant eux-mêmes l'information calculée) sont transmis au niveau du superviseur, de commande à commande par les mots-clés.

Dans une utilisation plus avancée de Python que la simple déclaration de commandes *Code_Aster*, le fichier de commande écrit en Python peut utiliser le contenu des structures de données propres à *Code_Aster*. En effet, Python peut être utilisé dans les fichiers de commandes pour créer des macro-commandes et des opérations comme des boucles (for, while, ...), des tests (if, ...), des exécutions de commandes externes (via le module os), etc… La page « Utilisation / Exemples / Exemples d'utilisation de Python dans Aster » du site web www.code-aster.org regroupe un certain nombre de cas d'application. Il est alors intéressant pour l'utilisateur de récupérer le produit des calculs fortran dans l'espace python, c'est-à-dire son fichier de commandes. Plusieurs méthodes Python ont été développées afin d'accéder au contenu d'autres structures de données.

Pour récupérer des données calculées (dans la mémoire JEVEUX), il est absolument nécessaire que les instructions entraînant leur obtention aient bien été exécutées au préalable. Autrement dit, il est indispensable d'exécuter le code en mode PAR_LOT='NON' (mot clé de la commande DEBUT ou POURSUITE). En effet, dans ce cas, il n'y a pas d'analyse globale du fichier de commande, mais chaque instruction est exécutée séquentiellement. Quand on arrive sur une instruction, tous les concepts la précédant ont donc déjà été calculés.

```
DEBUT ( PAR_LOT = 'NON')
```

Il faut alors noter que le fichier de commande ainsi produit n'est pas lisible par EFICAS qui ne tolère que les fichiers exclusivement composés de commandes propres à Code_Aster. Seules les variables simples (réels, entiers, strings) définies en mode déclaratif $a='toto'$ ou algébrique $n=3+4$ sont lisibles par EFICAS.

L'information relue dans la mémoire JEVEUX, produit d'un calcul préalable, peut être exploitée par exemple pour (liste non exhaustive) :

- Enchaîner conditionnellement d'autres commandes (exécution de boucle while jusqu'à obtention d'une valeur calculée de contrainte limite)
- Manipuler en python du contenu d'une table, d'une fonction, à fins de calculs
- Récupérer les attributs d'un maillage : liste des groupes de nœuds et de mailles, coordonnées.

2 Les tables

Les structures de données table sont produites dans Aster par création (CREA_TABLE), par lecture depuis un fichier (LIRE_TABLE) ou récupération dans un autre concept (RECU_TABLE). Ce sont fonctionnellement des tableaux de données hétérogènes (entiers, réels, chaînes de caractères) dont les colonnes sont identifiées par des noms de label.

Ce sont des structures pratiques dont l'emploi est généralisé dans le code. Par exemple, la plupart des commandes de post-traitement produisent des tables : pour relever des contraintes en des lieux géométriques donnés, pour produire des grandeurs macroscopiques calculées (post-traitements de mécanique de la rupture).

Soit par exemple la table tab1 suivante issue d'un calcul Aster :

NOEUD	NUME_ORDRE	DX
N2	14	0.93
N2	15	1.16
N1	3	0.70
N1	2	0.46
N1	1	0.23

Tableau 2-1

Elle aurait aussi pu être directement créée comme concept Aster de type table par :

```
tab1=CREA_TABLE(LISTE = (
          _F(    PARA='NOEUD',       LISTE_K=('N2','N2','N1','N1','N1'),),
          _F(    PARA='NUME_ORDRE',  LISTE_I=(14,15,3,2,1),),
          _F(    PARA='DX',          LISTE_R=(0 .93,1.16,0.70,0.46,0.23),),)
```

On peut directement récupérer une valeur quelconque de la table dont on connaît la clé d'accès (nom de label de colonne) et le numéro de ligne :

```
>>> print tab1['DX',3]
0.70
```

Il est aussi possible de récupérer la totalité de la table dans l'environnement python via une classe dédiée, produite par la méthode EXTR_TABLE, attachée à la classe du concept ASTER :

```
tab2 = tab1.EXTR_TABLE()
```

tab2 est un objet python, instance de la classe Table du module Utilitai.Table. Il est manipulable avec les méthodes associées à cette classe ; on pourra faire help(Table) pour connaître les méthodes de cette classe.

La table tab2 aurait aussi pu être définie directement par un dictionnaire :

```
From Utilitai.Table import Table
listdic = [  {'NOEUD': 'N2' ,'NUME_ORDRE': 14,'DX': 0.93, },
             {'NOEUD': 'N2' ,'NUME_ORDRE': 15,'DX': 1. 16,},
             {'NOEUD': 'N1' ,'NUME_ORDRE': 3 ,'DX': 0.70, },
             {'NOEUD': 'N1' ,'NUME_ORDRE': 2 ,'DX': 0.46, },
             {'NOEUD': 'N1' ,'NUME_ORDRE': 1 ,'DX': 0.23, },  ]
listpara=['NOEUD','NUME_ORDRE','DX']
listtype=['K8','I','R']
tab2=Table(listdic,listpara,listtype)
```

Les opérations possibles sur tab2 sont décrites ci-après.

2.1 Impression

```
>>> tab2
-------------------------------------------
NOEUD     NUME_ORDRE    DX
N2        14            9.30000E-01
N2        15            1.16000E+00
N1        3             7.00000E-01
N1        2             4.60000E-01
N1        1             2.30000E-01
```

Aussi possible :

```
>>> print tab2
```

Affichage d'un seul paramètre :

```
>>> t.DX
-------------------------------------------
DX
9.30000E-01
1.16000E+00
7.00000E-01
4.60000E-01
2.30000E-01
```

La commande IMPR_TABLE exploite les fonctionnalités d'impression offertes par cette classe. Le lecteur intéressé pourra lire la programmation python de cette macro-commande. En particulier la possibilité d'imprimer des tableaux croisés.

2.2 Création ou impression d'une sous-table extraite par filtre

Extraction suivant un seul critère :

```
>>> print tab2.NUME_ORDRE <=5
-------------------------------------------
NOEUD     NUME_ORDRE    DX
N1        3             7.00000E-01
N1        2             4.60000E-01
N1        1             2.30000E-01
```

Extraction suivant deux critères avec association logique « & » / ET :

```
>>> print (t.NUME_ORDRE < 10) & (t.DX>=0.3)
-------------------------------------------
NOEUD     NUME_ORDRE    DX
N1        3             7.00000E-01
N1        2             4.60000E-01
```

Extraction suivant deux critères avec association logique « | » / OU :

```
>>> print (t.NUME_ORDRE < 2) | (t.DX<0.5)
-------------------------------------------
NOEUD     NUME_ORDRE    DX
N1        1             2.30000E-01
N1        2             4.60000E-01
```

Extraction d'un nombre restreint de labels :

```
>>> t['DX','NUME_ORDRE']
-----------------------------------------
DX           NUME_ORDRE
9.30000E-01  14
1.16000E+00  15
7.00000E-01  3
4.60000E-01  2
2.30000E-01  1
```

Extraction suivant un critère d'égalité (ici avec valeur du critère déduite elle-même de la table)

```
>>> t.DX == max(t.DX)
-------------------------------------------
NOEUD     NUME_ORDRE    DX
N2        15            1.16000E+00
```

2.3 Tris

Tri de la table entière suivant un label :

```
>>> t.sort('NUME_ORDRE')
>>> t

-------------------------------------------
NOEUD     NUME_ORDRE    DX
N1        1             2.30000E-01
N1        2             4.60000E-01
N1        3             7.00000E-01
N2        14            9.30000E-01
N2        15            1.16000E+00
```

Pour trier selon plusieurs labels, l'ordre de préséance étant celui dans lequel sont déclarés les labels, il faut fournir les labels sous forme de liste ou de tuple :

```
>>> t.sort(['NUME_ORDRE',’DX’])
```

Un second argument `ordre`, valant `'CROISSANT'` ou `'DECROISSANT'`, permet de préciser l'ordre de tri :

```
>>> t.sort(['NUME_ORDRE',’DX’],'DECROISSANT')
```

2.4 Accès aux valeurs

Le contenu de la table est accessible par la méthode `values()` qui produit un dictionnaire dont les clés sont les paramètres d'accès de la table et les valeurs les colonnes :

```
>>> tab2.values()
{'NOEUD': ['N1', 'N1', 'N1', 'N2', 'N2'], 'NUME_ORDRE': [1, 2, 3, 14, 15],
'DX': [0.23, 0.46, 0.70, 0.93, 1.156]}
```

Les paramètres sont donnés par l'attribut `para` (idem `tab2.values().keys()`)

```
>>> tab2.para
['NOEUD', 'NUME_ORDRE', 'DX']
```

3 Méthodes d'accès sur les concepts

3.1 Accès aux contenu d'une SD liste

```
lst = [listr8] .Valeurs()
```

`lst` est une liste python qui contient les valeurs de la liste Aster : `lst = [ 0., 1.1, 2.3, ...]`

3.2 Accès au contenu d'une SD fonction ou d'une SD nappe

```
lst1, lst2 , (lst3) = [fonction / nappe] .Valeurs()
```

`lst1` et `lst2` sont deux listes python qui contiennent les abscisses et les ordonnées. Si la fonction est complexe, on obtient une troisième liste et `lst2` et `lst3` contiendront les listes des parties réelles et imaginaires.

```
lst1 = [fonction] .Absc()
```

`lst1` est la liste des abscisses, soit aussi la première liste renvoyée par `Valeurs()`.

```
lst2 = [fonction] .Ordo()
```

`lst2` est la liste des ordonnées, soit aussi la deuxième liste renvoyée par `Valeurs()`.

```
dico1 = [fonction] .Parametres()
```

retourne un dictionnaire contenant les paramètres de la fonction ; le type jeveux (`FONCTION`, `FONC_C`, `NAPPE`) n'est pas retourné, le dictionnaire peut ainsi être fourni à `CALC_FONC_INTERP` tel quel (voir efica02a).

3.3 Évaluation d'une SD fonction ou formule

Les fonctions et les formules sont évaluables simplement dans l'espace de nom python, donc le fichier de commandes, ainsi :

```
FONC1=FORMULE(   VALE='(Y**2)+ X',
                 NOM_PARA=('X','Y',)),
                 );
>>> print FONC1(1.,2.)
5.
```

ou avec une fonction :

```
FONC2=DEFI_FONCTION(  NOM_PARA='X',  VALE=(0., 0., 1., 4.,) )
>>> print FONC2(0.5)
2.
```

Dans le cas des fonctions, il faut noter qu'une tolérance de 1.e-6 en relatif est appliquée quand la valeur du paramètre se trouve très proche des bornes afin d'éviter une erreur pour cause de prolongement interdit à l'arrondi près.

3.4 Accès au contenu d'une SD maillage

Deux méthodes permettent de récupérer la liste des groupes de mailles et de nœuds d'une structure de donnée de type maillage :

```
[ (tuple), …]    = [maillage] .LIST_GROUP_MA()
```

renvoie une liste de tuples, chacun contenant le nom de chaque groupe de mailles, le nombre de mailles qu'il contient et la dimension (0, 1, 2 ou 3) la plus élevée de ses mailles :
`tuple = ('GMA', nb mailles, dim. mailles)`

`[ (tuple), ...]  =[maillage] .LIST_GROUP_NO()`

renvoie la liste des groupes de nœuds sous la forme :
`tuple = (nom du group_no, nb de nœuds du group_no)`

3.5 Accès au contenu d'une SD `matr_asse_*`

Soit `matr` une structure de données `matr_asse_depl_r`.

On récupère un tableau numpy de la matrice pleine en faisant :
```
array = matr.EXTR_MATR()
```

Pour récupérer la matrice avec un stockage creux, on fait :

```
data, lines, cols, dim = matr.EXTR_MATR(sparse=True)
```

On a :

`array` est de dimension `(dim, dim)`
nombre de termes non nuls : `len(data) = len(lines) = len(cols)`
`data[k] = array[ lines[k], cols[k] ]`

De même pour `matrgene` une structure de données `matr_asse_gene_r` , par exemple, produit e par l'opérateur `PROJ_MATR_BASE`.

On récupère un tableau numpy de la matrice pleine en faisant :
```
array = matrgene.EXTR_MATR_GENE()
```

3.6 Accès aux clés d'une SD `resultat`

Si `EVOL` est une structure de données `resultat`, alors :

`dico = EVOL.LIST_CHAMPS()`	est un dictionnaire dont les clés sont les noms des champs qui indexent la liste des numéros d'ordre calculés.

Tableau 3.6-1

Exemple :

`>>> print dico['DEPL']`	(le champ `DEPL` est calculé au numéros
`[0,1,2]`	d'ordre 0, 1 et 2)
`>>> print dico['SIEF_ELNO']`	(le champ n'est pas calculé)
`[]`	

Tableau 3.6-2

`dico = EVOL.LIST_VARI_ACCES()`	est un dictionnaire dont les clés sont les variables d'accès qui indexent leurs propres valeurs.

Tableau 3.6-3

Exemple :

`>>> print dico['NUME_ORDRE']`	(les numéros d'ordre du résultat `EVOL`
`[0,1,2]`	

	sont : 0, 1 et 2)
`>>> print dico['INST']` `[0., 2., 4.]`	(les instants calculés du résultat `EVOL` sont : 0.s, 2.s et 4.s)

Tableau 3.6-4

`dico = EVOL.LIST_PARA()`	est un dictionnaire dont les clés sont les paramètres du calcul qui indexent les listes (de cardinal égal aux nombre de numéros d'ordre calculés) de leurs valeurs.

Tableau 3.6-5

Exemple :

`>>> print dico['MODELE']` `['MO','MO','MO']`	(nom du concept modèle de référence pour chaque numéro d'ordre)
`>>> print dico['ITER_GLOB']` `[4,2,3]`	(nombre d'itérations de convergence pour chaque numéro d'ordre)

Tableau 3.6-6

3.7 Accès au contenu d' une SD `tran_gene`

Si `trangene` est une structure de données `tran_gene`, par exemple, produite par l'opérateur `DYNA_VIBRA`, alors :

`trangene.FORCE_AXIALE(inoli=-1)`	Matrice 1D du type « *numpy array* » contenant l'évolution de la force axiale aux instants archivés avec `inoli` étant l'indice de la non-linéarité, par défaut, `inoli` = -1 (première non-linéarité).
`trangene.FORCE_NORMALE(inoli=-1)`	Matrice 1D du type « *numpy array* » contenant l'évolution de la force normale aux instants archivés avec `inoli` étant l'indice de la non-linéarité, par défaut, `inoli` = -1 (première non-linéarité).
`trangene.FORCE_RELATION(inoli=-1)`	Matrice 1D du type « *numpy array* » contenant l'évolution de la force de la relation non-linéaire en déplacement ou vitesse aux instants archivés avec `inoli` étant l'indice de la non-linéarité, par défaut, `inoli` = -1 (première non-linéarité).
`trangene.INFO_NONL()`	Liste des informations des non-linéarités, y compris, l'indice et l'intitulé des non-linéarités.
`trangene.LIST_ARCH()`	Liste des instants archivés.
`trangene.VARI_INTERNE(inoli=-1)`	Matrice 2D type « *numpy array* » des toutes les variables internes de la non-linéarité donnée par l'indice <`inoli`>

Tableau 3.7-1

3.8 Accès au contenu d' une SD `melasflu`

Cette méthode python permet d'extraire la liste de vitesses du fluide pour lesquelles le calcul des paramètres de couplage fluide-élastique a été effectué.

```
>>> base = CALC_FLUI_STRU (...)

>>> print base.VITE_FLUI()
[1., 1.5, 2.5, 3.]
```

Liste de vitesses fluides pour lesquelles les coefficients de couplage ont été calculés.

3.9 Récupération en python des champs par éléments et champs aux nœuds (EXTR_COMP)

La méthode EXTR_COMP, appliquée à un champ, permet la récupération en python du contenu du champ.

3.9.1 Arguments de la commande EXTR_COMP

La commande possède 3 arguments :

ch1 = CHAMP.EXTR_COMP(comp=' ',lgma=[],topo=0), pour les champs aux noeuds,

ch1 = CHAMP.EXTR_COMP(comp,lgma,topo=0), pour les champs par élément,

comp	composante du champ sur la liste lgma. . Pour les champs aux noeuds, si comp est laissé par défaut, toutes les composantes sont retournées. Le résultat de la commande est modifié (voir ci-dessous).
lgma	liste de groupes de mailles, si vide alors on prend tous les group_ma (équivalent à TOUT='OUI' dans les commandes Aster.
topo	on renvoi des informations sur la topologie si >0 (optionnel, défaut = 0).

Tableau 3.9.1-1

Remarque : pour les champs aux nœuds, on peut lancer la commande de la manière suivante : ch1 = CHAMP.EXTR_COMP(topo=1). Dans ce cas, on retourne toutes les composantes pour toutes les entités topologiques du champs CHAMP.

3.9.2 Exemple d'utilisation

A partir du résultat U :

1) On crée un champ (noeud ou elXX) correspondant à un instant par CREA_CHAMP.
2) On extrait la composante par la méthode EXTR_COMP (déclarée pour les cham_elem et les cham_no) qui crée un nouveau type d'objet python : post_comp_cham_el et post_comp_cham_no dont les attributs sont décrits ci-après. On peut extraire toutes les composantes en une seule fois en ne précisant pas celle-ci (pour les champs aux noeuds uniquement).

```
U = STAT_NON_LINE( ... )

U104 = CREA_CHAMP(
       TYPE_CHAM   = 'NOEU_DEPL_R',
       OPERATION   = 'EXTR',
       RESULTAT    = U,
       NOM_CHAM    = 'DEPL',
       NUME_ORDRE  = 104,
       )

U104NP = U104.EXTR_COMP('DX',['S_SUP',])
```

```
print U104NP.valeurs


V104 = CREA_CHAMP(
       TYPE_CHAM   = 'ELGA_VARI_R',
       OPERATION   = 'EXTR',
       RESULTAT    = U,
       NOM_CHAM    = 'VARI_ELGA',
       NUME_ORDRE  = 104,
       )

V104NP = V104.EXTR_COMP('V22',[],1)

print V104NP.valeurs
print V104NP.maille
print V104NP.point
print V104NP.sous_point
```

3.9.3 Résultats de la commande EXTR_COMP

`ch1.valeurs : Numeric.array` contenant les valeurs

- Pour les champs par éléments, si on a demande la topologie (`topo>0`) :
  - `ch1.maille` : numéro de mailles
  - `ch1.point` : numéro du point dans la maille
  - `ch1.sous_point` : numéro du sous point dans la maille
- Pour les champs aux noeuds, si on a demande la topologie (`topo>0`) :
  - `ch1.noeud` : numéro des nœuds
  - `ch1.comp` : si on a demandé toutes les composantes du champs (`comp = ' '`,valeur par défaut), composante associée à la valeur.

4 Méthode d'accès à une structure de données quelconque

Il est possible de récupérer tout vecteur ou toute collection présente dans la mémoire, moyennant la connaissance de la structure de données.

Deux méthodes sont possibles (et équivalentes) : en utilisant le catalogue de la structure de données ou bien directement le nom JEVEUX de l'objet.

Dans le premier cas, on utilise la « propriété » `sdj` du concept qui permet de naviguer dans la structure de données (voir exemple ci-après).

4.1 Accès à une structure de données de type vecteur

La méthode `getvectjev` permet l'accès à une structure de données de type vecteur. Elle s'applique toujours sur l'objet « aster », et prend en argument la chaîne de caractère complète (espace y compris) définissant le nom de l'objet contenu dans la structure de donnée auquel on veut accéder. Celle-ci peut être déterminée grâce à la commande Aster `IMPR_CO (CONCEPT=_F( NOM=nom ))`.

Exemple : récupérer les coordonnées des nœuds d'un maillage nommé MA :

```
res = aster.getvectjev("MA      .COORDO    .VALE        ")
```

La syntaxe équivalente en utilisant le catalogue de structure de données est :

```
res = MA.sdj.COORDO.VALE.get()
```

On obtient une liste python contenant les valeurs du vecteur.

4.2 Accès à une structure de données de type collection

De manière analogue, la méthode `getcolljev` permet la consultation des collections depuis python. Elle renvoie un dictionnaire dont les clés sont les noms des objets en cas de collection nommée, les numéros d'indice sinon.

Exemple : récupérer les informations concernant la connectivité des éléments du maillage MA :

```
res = aster.getcolljev("MA      .CONNEX          ")
```

La syntaxe équivalente en utilisant le catalogue de structure de données est :

```
res = MA.sdj.CONNEX.get()
```

On obtient un dictionnaire ressemblant à :

```
{3: (2, 1, 5), 2: (6, 9, 10, 7, 11, 12, 13, 8), 1: (1, 6, 7, 2, 3, 8, 5)}
```