Introduire une nouvelle macro-commande

Résumé :

Ce document décrit comment définir et utiliser les macro-commandes en python.

Table des matières

1 Introduction

Ce document décrit l'utilisation et le développement des macro-commandes en Python pour Code_Aster. Ces macro-commandes peuvent être restituées dans le code et visibles de l'utilisateur comme des commandes à part entière. Mais elles peuvent aussi être placées dans le fichier de commande lui-même sans que l'utilisateur ait à toucher ni à l'exécutable, ni au catalogue de commandes. En outre, elles présentent l'avantage d'être écrites « naturellement » dans le langage de commande : le corps d'une macro-commande est semblable à un fichier de commandes ordinaire qui générerait la même séquence de commandes.

Le développeur aura grand intérêt à lire la programmation de macro-commandes existantes, sous `bibpyt/Macro` dans le répertoire d'installation de *Code_Aster* .

2 Qu'est-ce qu'une macro?

Une macro est une commande qui regroupe l'exécution de plusieurs sous-commandes. Elle est utilisable dans un fichier de commandes comme toute autre commande et possède son propre catalogue, définissant sa syntaxe. Plusieurs types de macro sont possibles :

- des macros propres au superviseur, implémentées en Python et en Fortran : par exemple `FORMULE` , `INCLUDE` , `DEBUT` ...
- des macros développeurs implémentées en Python.

Ce document traite de la définition et de l'utilisation des macros en Python.

3 Utiliser les macros

Utiliser les macros en Python est simple. Par rapport à des commandes simples comme des `OPER` ou des `PROC` , la seule différence porte sur les concepts produits par la macro. Une commande simple, de type `OPER` , a un seul concept produit que l'on trouvera à gauche du signe « = », comme suit :

```
concept = COMMANDE(mots-cles-simples-ou-facteurs)
```

Une commande simple de type `PROC` n'a aucun concept produit et s'écrit :

```
COMMANDE(mots-cles-simples-ou-facteurs)
```

Une macro-commande, de type `MACRO` , peut avoir plusieurs concepts produits. Un (mais ce n'est pas obligatoire) que l'on trouvera à gauche du signe =, comme pour un `OPER` , les autres comme arguments des mots-clés simples ou facteurs. On présentera le mode d'emploi pour un mot-clé simple. Il s'étend facilement aux mots-clés facteurs. Certains mots-clés sont susceptibles de produire des concepts. Pour demander à une macro commande de produire ce concept, l'utilisateur écrira à droite du signe = à la suite du nom du mot-clé, `CO` (' `nom_concept` ') , comme dans l'exemple qui suit :

```
ASSEMBLAGE(NUMEDDL=CO('num'))
```

Ceci a pour effet de créer un concept produit de nom `num` en sortie de la commande `ASSEMBLAGE` . Son type sera déterminé en fonction des conditions d'appel de la commande. `CO` est un nom réservé qui permet de créer des concepts produits nommés, non typés, préalablement à l'appel de la commande. C'est la commande qui attribuera le bon type à ce concept.

4 Définir une macro-commande en Python

Il faut définir :

- le catalogue proprement dit des mots-clés composant la macro,
- la méthode de typage des structures de données produites,
- la méthode Python définissant le corps de la macro : les commandes « de base » produites par la macro et leur enchaînement.

Les deux premiers points sont communs avec l'écriture d'une commande ordinaire (à l'exception d'une différence mineure dans la méthode de typage).

Il est possible de restituer tout ceci dans le catalogue de commandes du code et de rendre ainsi la macro-commande visible de tous. On peut également conserver son développement privé avec l'avantage de ne pas avoir à modifier les paramètres d'exécution ou l'exécutable en plaçant ces trois éléments directement en tête du fichier de commandes, ou en les important (import Python) depuis une localisation convenue.

4.1 Écrire le catalogue de la macro-commande

Le catalogue d'une macro est semblable à celui d'un commande simple. Les trois différences sont :

- on déclare un objet `MACRO` (et non `PROC` ou `OPER`),
- le mot clé réservé `op` ne contient pas un entier (désignant le numéro de routine FORTRAN de haut niveau pour `OPER` et `PROC`) mais un nom de méthode python,
- le concept produit n'est pas nécessairement unique, déclaré à gauche du signe « = ».

Des concepts produits peuvent être spécifiés en tant qu'arguments d'un mot clé simple. U n mot-clé simple qui définit un concept à produire est déclaré de type `CO` :

```
NUME_DDL = SIMP(statut='f',typ=CO)
```

Le type de l'objet retourné sera défini dans la fonction `sd_prod` (cf. paragraphe suivant).

Un même mot-clé peut avoir un double rôle : prendre un concept en argument ou produire un nouveau concept. Par exemple, on écrirait :

```
NUME_DDL = SIMP(statut='f',typ=(nume_ddl_sdaster, CO))
```

Cet usage est fortement déconseillé pour les développements futurs. Il est conseillé de définir un mot-clé pour chacune des fonctions.

4.2 Définir le type des concepts produits

La définition du type des concepts produits d'une macro-commande s'effectue de manière semblable à celle d'une commande simple de type `OPER` .

Si la commande ne produit qu'un concept que l'on trouvera à gauche du signe = comme dans :

```
a  =COMMANDE()
```

on procédera de la même manière que pour une commande simple de type `OPER` .

Dans le cas où la macro-commande peut produire plusieurs concepts dont certains en arguments de mots-clés, il faut ajouter quelques informations supplémentaires. Tout d'abord, il faut absolument

fournir une fonction Python, nommée sd_prod , dans la définition de la macro. Ensuite, les mots clés simples contenant le nom utilisateur du concept à produire doivent être de type CO (nom réservé).

Par exemple :

```
def ma_macro_prod(self, NUME_DDL, MATRICE,**args):
    if args.get('__all__'):
       return ([evol_noli], [nume_dll_sdaster], [None, matr_asse_depl_r])

   self.type_sdprod(NUME_DDL, nume_ddl_sdaster)
    if MATRICE:
      self.type_sdprod(MATRICE, matr_asse_depl_r)
   return evol_noli
....
MA_MACRO =MACRO(sd_prod=ma_macro_prod,...
                MATRICE  = SIMP(statut='f', typ=CO),
                ....
                NUME_DDL = SIMP(statut='o', typ=nume_ddl_sdaster),)
```

Dans ce cas, trois concepts peuvent être produits :

- de façon classique, un concept de type evol_noli qui aura été donné par l'utilisateur à gauche du signe « = ».
- un concept de type matr_asse_depl_r si l'utilisateur renseigne le mot clé simple MATRICE.
- un concept nume_ddl dont le nom est fourni au mot-clé NUME_DDL.

Lorsqu'un mot-clé peut avoir en argument un concept à produire, le mot-clé doit apparaître dans la liste des arguments de la fonction sd_prod et le concept doit être typé en utilisant la méthode type_sdprod de l'argument self qui est l'objet macro-commande.

NB : L'argument self n'est pas présent pour un OPER ou une PROC (il s'agit de l'objet MACRO).

Comme pour les commandes (cf. [D5.01.01]), la fonction sd_prod doit retourner l'ensemble des types possibles si __all__=True . Comme la macro-commande peut retourner plusieurs objets (3 dans l'exemple), si __all__=True , la fonction retourne une liste de 3 listes de types possibles, une liste pour chaque résultat.
Si un résultat n'est pas systématiquement produit (ici pour MATRICE), les types possibles sont alors None (rien n'est produit) et les véritables types.
S i la macro-commande ne retourne pas de résultat à gauche du signe « = », la première liste vaut alors [None] (à la place de [evol_noli] ici).
Dans certains cas, une macro-commandes retournent un nombre indéfini de résultats (un par occurrence d'un mot-clé facteur). Dans ce cas, on ne répète pas la liste correspondante.

5 Définir le corps de la macro

5.1 Organisation des modules

Dans le cas d'une macro privée, l'utilisateur organise ces modules Python comme il le souhaite.

En revanche pour une macro-commande intégrée au source officiel, il est nécessaire de respecter certaines règles.
Les fichiers sont au minimum :

- nom_macro.py : le catalogue de la commande
- nom_macro_ops.py : décrit du corps de la macro (en définissant la méthode principale par def nom_macro_ops(...))

Recommandations

- Dans `nom_macro.py` , il ne faut pas importer la méthode `nom_macro_ops` pour que le catalogue soit auto-portant (pour être utilisable avec uniquemnt le contenu de `code_aster/Cata`). On se contente de définir « l'adresse » vers la méthode, et précisant :
  `op=OPS('Macro.nom_macro_ops.nom_macro_ops')`
  La méthode définissant le corps de la macro-commande sera `nom_macro_ops` définie dans le fichier (module) `nom_macro_ops.py` du répertoire (package) `Macro` .
- La définition de la méthode `nom_macro_ops` doit respecter les conventions standards de codage en Python. Notamment, il ne faut pas faire une fonction de 1000 lignes : découper en fonctions élémentaires. De préférence, isoler ces fonctions élémentaires dans un autre module qui sera importé sous le corps de `nom_macro_ops` .

5.2 Transmission des mots clés de la macro à la méthode de construction (le corps)

Le corps de la macro sera défini dans une méthode de l'objet `MACRO` dont les arguments sont les arguments des mots-clés. Le premier argument est l'objet macro-commande, `self` , les suivants sont les mots-clés nécessaires pour exprimer le corps de la macro. Les mots-clés optionnels ou dont la présence est conditionnée par des blocs seront omis par l'emploi de l'argument `**args` .

Seuls les mots clés de haut niveau sont transmis : les MCSIMP de premier niveau, les MCFACT. Ces mots clés sont ensuite invoqués très simplement par leur nom.

Exemple de fonction corps :

```
def ma_macro_ops(self, UNITE_MAILLAGE, **args):
    ..................
          _nomlma = LIRE_MAILLAGE( UNITE = UNITE_MAILLAGE )
    ..................
```

Ici, `UNITE_MAILLAGE` est un MCSIMP de la macro, son contenu (concept, liste, string, … peu importe) est affecté au MCSIMP `UNITE` de la commande `LIRE_MAILLAGE` .

Cas d'un mot clé facteur :

```
def ma_macro_ops(self,MATR_ASSE_GENE,**args):
    ..................
          if MATR_ASSE_GENE.get('MATR_ASSE') is None:
    ..................
```

`MATR_ASSE_GENE` est un MCFACT de la macro, `MATR_ASSE` est un de ses sous-MCSIMP. Un MCFACT se manipule comme un dictionnaire.

Astuce : dans ce dernier exemple, on teste très simplement la présence de `MATR_ASSE` : si l'utilisateur n'a pas renseigné un mot clé (simple ou facteur), il n'est pas dans le dictionnaire et donc la méthode `get()` retourne `None` .

Remarque

Si un mot-clé simple n'accepte qu'une seule valeur (`max=1` dans sa définition), alors dans la macro-commande, ce mot-clé retourne la valeur renseignée. Par exemple, `INFO` retourne 1 ou 2 (un unique entier).
Si un mot-clé simple est défini avec `max>1` ou `max='**'` , alors la valeur retournée sera toujours un tuple même si l'utilisateur n'a fourni qu'une valeur. Par exemple, `GROUP_MA` (souvent défini avec `max='**'`) retournera `('GM1', 'GM2')` ou `('GM1', )` s'il n'y a qu'une valeur.

Pour un mot-clé facteur, on retourne toujours une liste.

5.3 Appeler une commande dans le corps de la macro

Pour appeler une commande dans le corps de la macro, il faut importer son catalogue (regrouper tous les imports ensemble) :

```
from code_aster.Cata.Commands import NUME_DDL
num = NUME_DDL(METHODE=...,...)
```

Il n'y a aucun obstacle à ce que les commandes « filles » produites par la macro-commande soient elles-même des macro-commandes.

Si on définie une FORMULE dans le corps d'une macro-commande et que dans l'expression de celle-ci on utilise des constantes (par exemple a dans `VALE='''a*INST'''`), il faut en déclarer la valeur explicitement avant d'évaluer la formule :

```
self.update_const_context({ 'a' : 2. })
... puis CALC_FONC_INTERP ou autre...
```

5.4 Nommage des concepts produits dans le corps par la macro

Les concepts produits dans le corps de la macro sont de plusieurs sortes :

- les concepts nommés et détruits automatiquement à la fin de l'exécution de la macro-commande. Il ne faut donc plus en avoir besoin par la suite et il faut en particulier veiller à ce que le concept produit par la macro n'y fasse pas référence. Pour indiquer qu'il s'agit d'un concept de cette sorte, il suffit de lui donner un nom qui commence par __ (double underscore)

  Exemple :

  ```
  __a=CALC_MATR_ELEM(MODELE=MODELE)
  ```

  Alors, comme on peut le lire dans le fichier de messages, un nom de concept automatique, précédé d'un point est généré :

  ```
  .9000005=CALC_MATR_ELEM(MODELE=MODELE)
  ```

  Une fois sorti de la macro, l'objet correspondant au nom de concept `.9000005` n'existe plus, ni dans l'espace de noms du superviseur, ni dans la base `jeveux` .

- les concepts nommés automatiquement et conservés dans la base `jeveux` à la fin de la macro-commande. Pour indiquer qu'il s'agit d'un concept de cette sorte, il suffit de lui donner un nom qui commence par _ (simple underscore)

  Exemple :

  ```
  _a=CALC_MATR_ELEM(MODELE=MODELE)
  ```

  Alors, comme on peut le lire dans le fichier de messages, un nom de concept automatique, précédé d'un underscore est généré :

  ```
  _9000005=CALC_MATR_ELEM(MODELE=MODELE)
  ```

Une fois sorti de la macro, l'objet correspondant au nom de concept _9000005 n'existe pas dans l'espace de noms du superviseur, il est en revanche présent sous ce nom dans la base jeveux.

Ce type d'objet répond aux situations particulières où le concept produit par la macro « dépend » d'un concept amont qui devra être toujours présent : par exemple un modele relativement à un maillage, une matr_asse relativement à un nume_ddl.

- les concepts destinés à devenir les concepts produits de la macro. Pour indiquer qu'il s'agit d'un concept de cette sorte, il faut appeler la méthode DeclareOut de l'objet macro self avec, comme arguments, le nom de la variable de retour de la commande et l'objet issu des mots-clés de la macro-commande.

Exemple : on associe la variable locale mm au mot-clé MATRICE de la macro:

```
self.DeclareOut('mm', MATRICE)
mm=ASSE_MATRICE(.............)
```

- le concept de sortie de la macro (nécessairement unique) est traité de manière semblable en indiquant le concept de sortie self.sd.

```
self.DeclareOut('mm', self.sd)
mm=ASSE_MATRICE(.............)
```

mm deviendra le concept de sortie de la macro, il portera le nom donné par l'utilisateur dans son fichier de commandes (et non mm).

5.5 Récupération des exceptions dans une macro

Le fonctionnement du superviseur étant particulier et différent selon qu'une commande est exécutée dans le jeu de commandes ou dans une macro-commande, le traitement en cas d'exception doit lui aussi est particulier.
Voyons cela sur un exemple dans le jeu de commandes :

```
try:
. . resu = STAT_NON_LINE(...)
except NonConvergenceError as exc:
. . /code/
```

Dans /code/, on a accès à resu. Pourquoi ? parce que dès l'appel à la commande, resu est placé dans jdc.g_context et que le contenu du jeu de commandes est exécuté dans ce même contexte.

Si on a besoin de faire ce type de récupération dans une macro, ça ne fonctionne pas de cette manière, c'est plus conforme à du code Python ordinaire.
Il faut faire appel à une méthode dédiée à cela : get_last_concept(). Celle-ci permet de récupérer la coquille assd sur le dernier concept produit dans cette macro.
On fera alors :

```
try:
. . __resu = STAT_NON_LINE(...)
except NonConvergenceError as exc:
. . resu = self.get_last_concept()
. . /code/
```

__resu est le concept temporaire créé dans la macro (sous par exemple le nom jeveux .9000027).

resu est un objet assd qui permet d'accéder à la structure de données fortran de nom .9000027. Dans ce cas, resu.List_Vari_Acces() fonctionnera.

Si la commande émet une erreur 'F', on récupérera quand même un objet `assd` mais la structure de données fortran n'existera pas car on aura vu le message "`Destruction du concept '.9000027'`" (d'où l'intérêt de ne récupérer que les exceptions typées, comme dans l'exemple, sinon il est très difficile de savoir dans quel état est le concept).

Dans les exemples ci-dessus, la variable `exc` contient l'objet dérivé de `aster.error`.
Cet objet dispose d'attribut permettant de savoir quelle exception a été levée et avec quel identifiant de message.

5.6 Accéder aux concepts produits avant la macro

D'une manière générale, tous les concepts nécessaires au fonctionnement de la macro doivent transiter par ses mots-clés.

Dans quelques cas particuliers, il est nécessaire de récupérer un concept dont on ne connait que le nom. Par exemple, le nom est stocké dans une structure de données en entrée. Pour obtenir le concept à partir d'un nom, il faut utiliser la méthode `get_concept` de l'objet `MACRO`. Soit dans le corps de la macro :

```
objet = self.get_concept(nom)
```

Dans le même ordre d'idée, on peut avoir besoin de l'ensemble des objets disponibles (par exemple, `STANLEY` propose le choix parmi les concepts déjà calculés). Pour cela, on récupère le contexte comme ceci :

```
dictionnaire = self.get_contexte_courant()
```

La méthode `get_concept` donne accès aux concepts enregistrés auprès du jeu de commandes. Un concept temporaire créé dans une macro-commande mais qui a été caché à l'utilisateur (nommé par exemple `_9000028`), ne peut être récupéré directement. Il faut créer une référence vers la structure de données fortran en utuilisant `get_concept_by_type`. Exemple :

```
objet = self.get_concept_by_type('_9000028', maillage_sdaster)
```

NB : l'utilisation du dictionnaire `sds_dict` est à proscrire (pour les deux tâches précitées), il s'agit d'un objet interne au superviseur.

5.7 Numéroter la macro

Dans les affichages du fichier de messages, toutes les commandes sont numérotées. Pour incrémenter ce compteur, il faut systématiquement appeler la méthode suivante en tête du corps de la macro :

```
self.set_icmd(1)
```

Il est particulièrement important de faire cet appel avant toute commande « fille » de la macro commande car cette méthode initialise également la mesure de temps CPU globale pour la macro.

5.8 Traiter les erreurs

Comme pour une commande en Fortran, il est possible de détecter des erreurs d'utilisation dans le corps de la macro par l'utilitaire Utmess, identique dans son fonctionnement à son homonyme FORTRAN ([D6.04.01] - Utilitaires d'impression de messages). Pour ce faire, il faut importer cette méthode depuis le module des utilitaires Python.

Exemple :

```
from Utilitai.Utmess import UTMESS
...
UTMESS('F', 'CHARGES_4', valk="DX", vali=(2, 88))
```

Le premier argument indique la nature de l'erreur ou alarme, le second précise l'identifiant du message dans le catalogue de messages et les arguments suivants (`vali`, `valr`, `valk`) permettent de fournir des variables entières, réelles ou chaines de caractères pour compléter le message.

5.9 Les affichages

Autant que cela est possible, les messages à destination de l'utilisateur seront affichés avec `UTMESS('I',...)` . Cela permet de définir des messages cohérents et qui pourront être traduits automatiquement.
Quand il n'est pas possible d'utiliser le catalogue de messages, il est possible d'imprimer directement du texte sur le fichier `MESSAGE` ou `RESULTAT` en employant la méthode `affiche` du module `aster` .
L'utilisation de la commande `print` est fortement déconseillée.

Exemple d'utilisation de `aster.affiche` :

```
import aster
…
aster.affiche('MESSAGE', chaine_de_caracteres)
```

Le premier argument vaut ' `MESSAGE` ' ou ' `RESULTAT` ' suivant le fichier cible.

5.10 Identifier les concepts produits par la macro

Dans certaines circonstances, il faut déterminer si un concept est produit par la macro elle-même ou a été produit par une commande précédente. Ceci est possible en testant si le concept en question est présent ou non dans la liste des concepts produits par la macro qui est donnée par l'attribut `sdprods` de l'objet macro self.

Exemple :

```
if nume_ddl in self.sdprods :
   # le concept nume_ddl est produit par la macro
   # il faut appeler la commande NUME_DDL
   lnume = 1
else:
   # le concept nume_ddl existe déjà.
   lnume=0
```

Attention, `sdprods` ne contient pas le concept produit retourné par la macro qui se trouve dans l'attribut `sd` de `self` .

5.11 Création dynamique de commandes : nombre de mots-clés variables, contenu contextuel

Dans certains cas, suivant la valeur des options, une même commande sera appelée avec différents mots-clés ou des arguments différents. Pour traiter cette situation et générer dynamiquement la commande, on construit un dictionnaire contenant les mots clés à écrire qui est ensuite transmis en argument de la commande, précédé des caractères '**'. Le dictionnaire est alors « déplié », les clés sont les arguments (mots clés), suivis du contenu de la clé, derrière le signe « = ».

Ce dictionnaire Python peut être construit au fur et à mesure de l'examen des options.

Exemple :

```
moscles={}
moscles['INFO'] = 2

motscles['CREA_GROUP_NO'] = []
for grma in GROUP_MA_BORD :
    motscles['CREA_GROUP_NO'] . append( _F(GROUP_MA = grma) )

_nomlma = DEFI_GROUP (  reuse = _nomlma
                                        MAILLAGE = _nomlma,
                                        **motscles )
```

Le dictionnaire `motscles` contient la définition de `INFO` et une liste de mots-clés facteurs `CREA_GROUP_NO` . Pour l'exemple, la liste est une simple recopie de `GROUP_MA_BORD` construite par ajout successifs d'un élément.

5.12 Appel à un code externe

Si on souhaite exécuter dans la macro un code tiers par la commande `EXEC_LOGICIEL` ou en utilisant le module `Utilitai.System` , on doit récupérer le chemin vers le logiciel externe. S'il s'agit d'un logiciel officiellement supporté, son chemin d'accès est enregistré dans un fichier de l'installation nommé `external_programs.js`. On le récupère en faisant :

```
import aster_core
miss3d = aster_core.get_option('prog:miss3d')
EXEC_LOGICIEL( ... , LOGICIEL=miss3d ,  ... )
```

5.13 Règles de programmation

Le respect de règles de programmation est nécessaire pour améliorer la lisibilité et la cohérence du code. L'absence de vérification de ces règles de programmation conduit à une grande disparité du code Python de *Code_Aster* ce qui rend difficile la compréhension rapide de celui-ci.

Il faudrait impérativement de conformer aux préconisations de la PEP 8 – Style guide for Python code : http://www.python.org/dev/peps/pep-0008/

Quelques règles simples à respecter :

- Indentations : 4 espaces (et non 2 ou 3), pas de tabulations.
- Largeur maximale de 80 caractères (jusqu'à 90-100, c'est tolérable, plus cela devient vraiment illisible).
- Une seule instruction par ligne (pas d'instruction sur la même ligne que `if/else`).
- Un seul `import` par ligne, jamais de '`import *`'.
- Ne pas multiplier les lignes blanches de séparation : deux entre les fonctions/classes au plus haut niveau, une à l'intérieur d'une classe.
- Un espace entre les opérateurs (+, -, *...), après la virgule, après ' :' facilite la lecture.
- La chaîne doc des fonctions est obligatoire pour décrire ce qu'elles font.
- Nommer les variables en minuscules, les classes avec une majuscule à chaque mot...

En particulier pour *Code_Aster* :

- En un coup d'oeil, on doit comprendre ce que fait une macro ou une fonction : découper le code en tâches élémentaires de 50-100 lignes chacune. Séparer en plusieurs fichiers pour que cela reste lisible (<500 lignes).
- Les mots-clés étant en majuscules, utiliser les minuscules pour les variables.
- Le nommage des concepts est suffisamment compliqué pour prendre soin de les nommer en minuscules pour bien les distinguer des mots-clés. Sauf les concepts produits, ils

commencent par un ou 2 '_'. Ne pas nommer des variables Python ordinaires en commençant avec un '_'.

Outils : `reindent.py` fournit avec Python permet de réindenter du code selon la convention, `pylint` fait un diagnostic du respect des conventions...

6 Considération sur l'usage des macros en Python

6.1 Définition d'une macro hors catalogue

La méthode standard pour ajouter la définition d'une macro en Python pour une exécution de *Code_Aster* est de l'ajouter dans le catalogue de référence du code.
Cependant, dans certains cas :

- macro personnelle,
- test lors du développement,

il peut être pratique d'ajouter la définition de la macro en dehors du catalogue. Pour ce faire, il suffit de créer un module Python contenant la définition de la macro en ajoutant en tête du module l'import des variables du catalogue.

Exemple simplifié :

```
from code_aster.Cata.Syntax import ...
from code_aster.Cata.DataStructure import ...
def ma_macro_prod(self,...):
    .....
def ma_macro_ops(self,...):
....
MA_MACRO=MACRO(nom='MA_MACRO',...)
```

Puis à l'utilisation, il suffit dans le fichier de commandes de faire l'import de la macro précédemment définie.

Exemple de fichier de commandes :

```
# la macro MA_MACRO est définie dans le module ma_macro.py
from ma_macro import MA_MACRO
a=MA_MACRO(...)
```

Il est aussi possible d'utiliser la fonctionnalité d'`INCLUDE` :

```
# la macro MA_MACRO est définie dans le fichier INCLUDE 45
INCLUDE(UNITE=45)
a=MA_MACRO(...)
```