pyexp

Service Informatique Expériences - L.U.R.E.

pyexp

par Service Informatique Expériences - L.U.R.E.

PyExp - Utilisation

Laurent Pointal
Informaticien, développeur d'applications
C.N.R.S.

PyExp - Utilisation
par Laurent Pointal

Vu du côté utilisateur, PyExp permet d'écrire assez simplement des scripts pour exprimer des logiques expérimentales puis les exécuter. Il aborde l'expérience en terme de manipulation de grandeurs physiques, et offre des fonctions qui permettent de réaliser des opérations sur des ensembles de grandeurs sans avoir à connaître à priori la façon dont ces manipulations se traduisent au niveau des matériels constituant l'expérience.

Table des matières

Liste des tableaux

Liste des illustrations

Chapitre 1. Introduction

PyExp se base sur une description de la configuration du poste expérimental (voir Chapitre 3) qui identifie par des noms :

- chacune des grandeurs physiques que l'on peut avoir à manipuler,
- les matériels présents sur l'expérience,
- les équipements au fonctionnement plus complexe (monochromateurs, goniomètres, fentes pilotées…) résultant généralement de la coordination de plusieurs matériels.

Cette description contient toutes les informations relatives aux paramètres de fonctionnement des grandeurs, matériels et équipements, ainsi que les informations concernant les relations qu'ils ont entre eux (par exemple que la grandeur nommée `Theta` est liée au 3ème axe du tiroir de contrôle d'axes nommé `Moteurs`). Les scripts basés sur PyExp manipulent des objets correspondant à cette description afin de réaliser les opérations nécessaires au déroulement de l'expérience.

Vocabulaire

Dans PyExp, les grandeurs, matériels et équipements sont tous à la base des acteurs (objets **`Actor`**) qui collaborent entre eux. Les matériels sont plus spécifiquement des objets **`Device`**, les équipements des objets **`Group`** et les grandeurs physiques des objets **`PhysElement`**.

Petit exemple

A partir de la ligne de commande de Python, il est possible de démarrer directement PyExp en l'important.

```
>>> from pyexp import *
================================================================================
pyexp - V0.3.4 alpha 0 - 2002-08-21

copyright 2001-2002 CNRS/CEA - LURE
license GPL V2 or later

    To get help on a function or object or class x, use: help (x)
================================================================================
Starting pyexp...
Opening syslog for you as 'testenligne' application, and with console log!!!
Experiment name: SurCrocus
Experiment application name: CommandLine
User profile: Anonymous
Experiment configuration source URL: fileconf:O:\configs\manip.cfg
Start finished. Ready.
================================================================================
```

On trouve alors le prompt du shell Python avec l'environnement de PyExp chargé, et on peut directement manipuler les grandeurs physiques.

```
pyexp >>> l=getactor("E")
pyexp >>> print l
<PhysElement E>
pyexp >>> set(l,250)
```

```
pyexp >>> get(l)
[249.99999999999997]
pyexp >>> c1,c2=getactor("C1"),getactor("C2")
pyexp >>> measure(2.5,c1,c2)
pyexp >>> get(c1,c2)
[25.0, 50.0]
pyexp >>> f=open("data.txt","w")
pyexp >>> save(f,c1,c2)
pyexp >>> f.close()
```

Ici on a simplement modifié la longueur d'onde sur l'expérience, mesuré sur deux compteurs durant deux secondes et demi, et enfin sauvegardé les valeurs mesurées. Le fichier résultant contient simplement :

```
25     50
```

Nous allons, dans la suite de l'introduction, faire un tour rapide des grandeurs et lister les différentes fonctions et méthodes couramment utilisées dans les scripts expérimentaux.

Liens entre acteurs

Pour bien utiliser PyExp, il est important de comprendre comment les grandeurs physiques, les goupes et les devices sont liés entre eux (les détails concernant la façon dont ils communiquent sont présentés dans le document *PyExp - Machinerie*).

Les slots

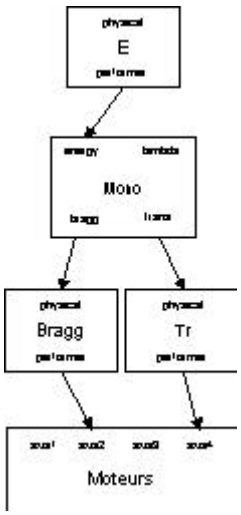
Chaque objet acteur de PyExp dispose d'un ou plusieurs slots. Ce sont des ports de communication nommés qui permettent aux acteurs de recevoir ou d'envoyer des demandes d'actions. On distingue les **upslots** qui servent normalement à recevoir des demandes d'action (en provenance d'autres acteurs ou en provenance des fonctions et méthodes de haut niveau), et les **downslots** qui servent normalement a envoyer des demandes d'actions (vers d'autres acteurs). Exemples :

- Un acteur grandeur physique E (lié au réglage de l'énergie sur l'expérience) avec un upslot `physical` et un downslot `performer`.
- Un acteur groupe Mono (chargé de gérer la logique de fonctionnement d'un monochromateur comportant un axe de rotation et un axe de translation) avec deux upslots `energy` et `lambda`, et deux downslots `bragg` et `trans`.
- Un acteur grandeur physique Bragg (lié au réglage de l'angle de Bragg sur le monochromateur) avec un upslot `physical` et un downslot `performer`.
- Un acteur grandeur physique Tr (lié au réglage de la translation (hauteur) de l'expérience en sortie du monochromateur) avec un upslot `physical` et un downslot `performer`.
- Un acteur device Moteurs (chargé de gérer un périphérique contrôlant physiquement 4 moteurs) avec quatre upslots `axis1`, `axis2`, `axis3` et `axis4`.

Les liens

A partir des acteurs et de leurs slots, il est possible de tisser des liens. Chaque lien part d'un downslot d'un acteur et arrive à un upslot d'un autre acteur. On crée ainsi un graphe d'acteurs représentant l'expérience pilotée.

Figure 1-1. Exemple simple de liens entre acteurs



Grandeurs

Les objets grandeurs physiques sont les principaux acteurs de PyExp que vous aurez à manipuler directement. Ils permettent d'accéder aux valeurs associées aux grandeurs (lecture et/ou modification), d'accéder aux indications d'unité, de sauvegarder les valeurs dans des fichiers, de lire/modifier des paramètres liés aux grandeurs, d'effectuer des mesures… Les valeurs associées aux grandeurs peuvent être de tous types (flottant, entier, tableau…).

Pour manipuler les grandeurs PyExp met à votre disposition des fonctions et des méthodes. Une **méthode** s'applique à un objet spécifique, en précisant l'objet avant l'appel de la méthode, par exemple : `e.set(12000)`. Une **fonction** s'appelle directement, par exemple : `set(e,12005)`. PyExp définit certaines méthodes comme `set` sous la forme de fonctions homonymes offrant les mêmes fonctionnalités… mais permettant de manipuler plusieurs grandeurs physiques simultanément.

Unités et conversions

Les objets grandeurs physiques donnent accès aux valeurs associées aux grandeurs dans deux unités possibles : une **unité utilisateur** et **une unité data**. La première est l'unité normale que vous manipulez couramment. La seconde est l'unité que doit utiliser l'objet grandeur pour piloter la grandeur physique. Ceci permet par exemple d'avoir une grandeur liée à une position angulaire, exprimant l'angle en radians pour l'utilisateur, et donnant cette valeur en degrés pour le pilotage de l'axe concerné. Ou encore d'avoir une unité utilisateur en millimètres alors que les valeurs fournies au périphérique doivent être exprimées en microns.

Un système de conversion (voir Chapitre 2 dans *PyExp - Machinerie*) est associé à chaque objet grandeur afin de passer d'une unité à l'autre.

Méthodes d…

> Les méthodes qui permettent de s'exprimer en unité utilisateur ont généralement une méthode équivalente exprimée en utité data et préfixée par 'd'. Par exemple il existe des méthodes `get` et `dget`, `read` et `dread`, `unit` et `dunit`...

Informations sur les unités. Ces méthodes permettent de connaître les unités liées à la grandeur : `unit`, `unitverb`, `dunit`, `dunitverb`.

Conversions entre unités. Ces méthodes permettent d'effectuer des conversions de valeurs entre unités avec les paramètres de la grandeur : `u2d`, `d2u`, `user2data`, `data2user`, `u2dforset`, `absvalue`, `relvalue`, `absdvalue`, `reldvalue`.

Fixation des valeurs

Certaines méthodes et fonctions permettent de travailler avec des valeurs relatives (par rapport à la valeur courante), elle sont préfixées par `r`. Les méthodes suivantes permettent de modifier la valeur associée à la grandeur physique (avec généralement un effet concret sur du matériel, comme un mouvement sur un moteur ou l'ajustement d'une tension de sortie sur un convertisseur numérique-analogique) : `set`, `dset`, `rset`, `rdset`. Des fonctions générales de PyExp permettent de fixer les valeurs de plusieurs grandeurs physiques à la fois : `set`, `rset`, `dset`, `rdset`.

Mesures

Dans PyExp une mesure est la réalisation de l'intégration d'une valeur pendant un certain temps, via un système électronique ou numérique. On distingue la mesure (qui peut prendre du temps) de la lecture (qui peut dans certains cas être quasi immédiate). Certains appareils ne permettent une lecture qu'après avoir effectué une mesure.

Mesure des valeurs. Les méthodes suivantes permettent de gérer des mesures sur une grandeur physique : `measure`, `premeasure`, `postmeasure`. Des fonctions générales de PyExp permettent de gérer les mesures sur plusieurs grandeurs physiques à la fois : `measure`, `premeasure`, `postmeasure`.

Accès aux valeurs

Rafraichissement des Valeurs

> Chaque objet grandeur de PyExp mémorise la dernière valeur associée à la grandeur, et maintient un indicateur permettant de savoir si cette valeur est à jour. Ceci permet de n'effectuer les lectures sur les appareils que lorsque cela est nécessaire. Normalement **on utilise les méthodes `get`** (lecture si besoin) **plutôt que les méthodes `read`** (lecture forcée). De plus certains objets device de contrôle d'appareils effectuent automatiquement une lecture en fin de mesure - et ne permettent pas de forcer une nouvelle lecture n'importe quand - voir ne supportent pas du tout la demande de lecture forcée.

Récupération des valeurs courantes. Les méthodes suivantes permettent d'accéder à la valeur courante associée à la grandeur physique, en lançant des lectures uniquement si la valeur stockée dans la grandeur n'est pas à jour : `get`, `dget`. Des fonctions générales de PyExp permettent de récupérer les valeurs de plusieurs grandeurs physiques à la fois : `get`, `getasdict`, `dget`, `dgetasdict`.

Récupération des valeurs précédentes. Les méthodes suivantes permettent d'accéder à la valeur précédemment associée à la grandeur physique : `getlast`, `dgetlast`.

Récupération des valeurs avec lecture forcée. Les méthodes suivantes permettent d'accéder à la valeur associée à la grandeur physique, en en forçant la lecture même si la valeur stockée est considérée comme étant à jour : `read`, `dread`. Des fonctions générales de PyExp permettent de lire plusieurs grandeurs physiques à la fois : `read`, `readasdict`, `dread`, `dreadasdict`.

Sauvegarde des valeurs

Sans imposer de solution de stockage spécifique, PyExp propose un système permettant le stockage dans des fichiers des valeurs associées aux grandeurs physiques (valeurs formatées suivant le type de donnée associée à chaque grandeur et suivant des paramètres de formatage spécifiables pour chaque grandeur). Les méthodes suivantes permettent de sauvegarder simplement la valeur associée à la grandeur physique dans un fichier : `save`, `dsave`. Des fonctions générales de PyExp permettent de sauvegarder plusieurs grandeurs physiques à la fois : `save`, `dsave`.

Paramétrage et actions

En plus de pouvoir contrôler la valeur associée aux grandeurs, il est possible de gérer les paramètres de fonctionnement des grandeurs, par exemple modifier la vitesse pour une grandeur physique axe liée au pilotage d'un moteur. Chaque paramètre est nommé (nom en majuscules), le paramètre par défaut `VALUE` correspondant à la valeur associée à la grandeur physique (c'est le paramètre contrôlé par les get, set, read…). Pour notre exemple le paramètre contrôlant la vitesse d'un axe est `SPEED`.

Lorsque l'on désire réaliser une action standard sur une grandeur, on a généralement une méthode ou une fonction qui permettent de le faire simplement, par exemple pour réaliser une action `SET`, on dispose des méthodes et fonctions `set`, `dset`, `rset`, `rdset`. Mais il peut arriver que certaines fonctionnalités d'un device ou d'un groupe ne soient pas accessibles par ce biais, PyExp offre des méthodes et fonctions pour demander l'exécution d'actions en donnant directement leur nom (en majuscules) et les paramètres nécessaires.

Accès aux paramètres. Les méthodes suivantes permettent de manipuler les paramètres des grandeurs physiques : `setparam`, `getparam`, `getoneparam`. De plus les grandeurs héritent des méthodes d'accès aux paramètres définies pour tous les acteurs, et indépendantes des noms de slots : `setslotparam`, `getslotparam`, `getslotoneparam` ; et les grandeurs physiques peuvent être utilisées avec les fonctions générales de PyExp correspondant à ces méthodes : `setslotparam`, `getslotparam`, `getslotoneparam`.

Par ailleurs une méthode `infos` permet de récupérer, sous forme d'un texte directement affichable, une synthèse des informations relatives à la grandeur (toutes ses caractéristiques).

Lancement d'actions. La méthode suivante permet de demander l'exécution d'une action sur une grandeur physique : `execute`. Les fonctions de haut niveau de PyExp permettent aussi de lancer des actions sur une ou sur plusieurs grandeurs physiques en même temps : `execute`, `executemulti`. De plus les grandeurs physiques héritent de la méthodes de lancement d'action définie pour tous les acteurs indépendamment des noms de slot : `executeslot`. Enfin les grandeurs physiques peuvent être utilisées avec les fonctions de haut niveau de PyExp pour les lancements d'actions sur des acteurs quelconques : `executeslot`, `executemultislot`.

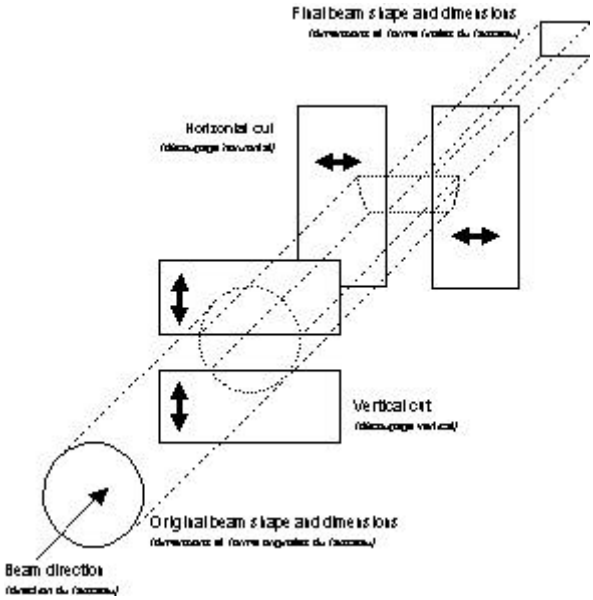
Autres acteurs notables

Les grandeurs physiques sont les principaux objets que vous aurez à manipuler dans les scripts expérimentaux. Mais elles sont liées à d'autres objets, groupes ou devices qui apportent différentes fonctionnalités. Nous allons lister ici certains de ces acteurs

Systèmes de fentes

Une fente (slit) est un système mécanique composé de deux lèvres (blade) mobiles permettant de découper en largeur ou bien en hauteur un faisceau de lumière. Pour pouvoir être pilotée, une fente doit être équipée de moteurs

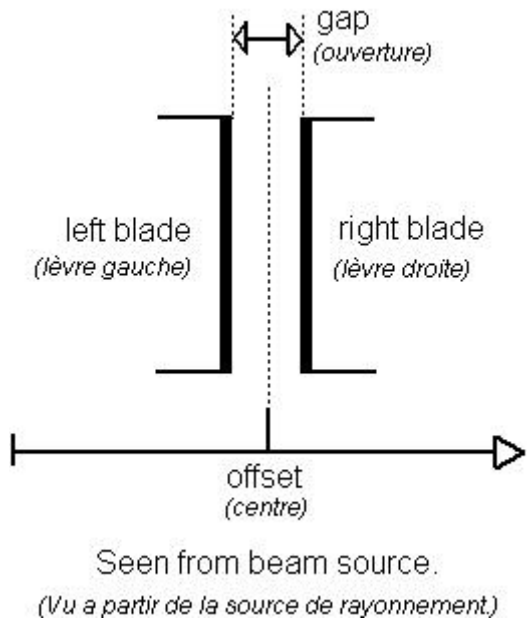
Figure 1-2. Utilisation d'un système de fentes



Les groupes de fentes (slitset) sont des intermédiaires chargés de gérer un ou plusieurs exemplaires de fentes et d'en présenter un fonctionnement standard d'une fente modèle. La fente modèle permet de piloter l'ouverture (gap) et le centrage (offset) du passage de la lumière.

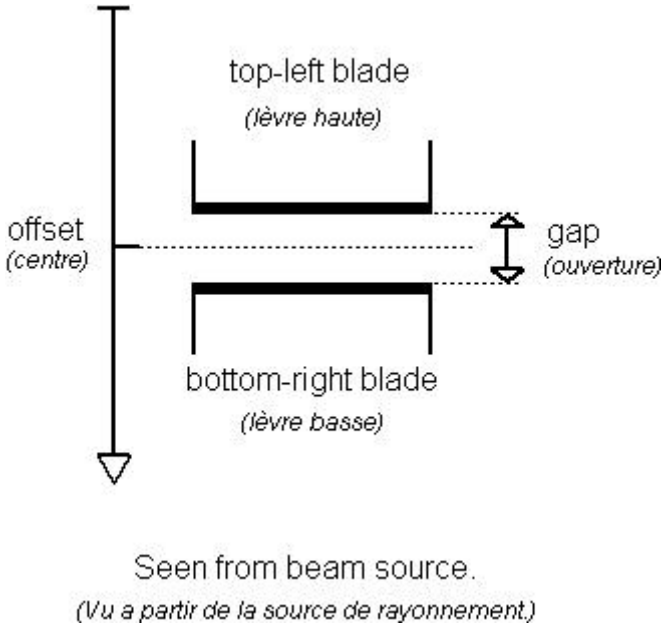
Pour une fente à découpage horizontal, le modèle est le suivant :

Figure 1-3. Modèle de fente à découpage horizontal



Et pour une fente à découpage vertical, le modèle est le suivant (idem modèle à découpage horizontal avec un quart de tour dans le sens des aiguilles d'une montre) :

Figure 1-4. Modèle de fente à découpage vertical



Lorsque l'ouverture croît, les lèvres s'écartent. Lorsque le centrage croît, la fente se déplace vers la droite/le bas. Le groupe se charge de transformer les requêtes concernant l'ouverture et le centrage en des requêtes correspondantes au niveau des moteurs pilotant les lèvres.

Chapitre 2. Exemples commentés

Nous allons donner un premier exemple d'utilisation en nous basant sur une simulation d'expérience avec les acteurs suivants : Vous pouvez voir le fichier de description correspondant à cette configuration dans Annexe A. Le résultat de l'exécution du script:

```
...
Start finished. Ready.
================================================================================
['E',
 'Trsin',
 'Levre1G',
 'Levre1D',
 'Levre2H',
 'Levre2B',
 'Levre3G',
 'Levre3D',
 'Levre4H',
 'Levre4B',
 'F1gap',
 'F1ofs',
 'F2gap',
 'F2ofs',
 'F3gap',
 'F3ofs',
 'F4gap',
 'F4ofs',
 'I1',
 'I2',
 'I3',
 'I4',
 'I5',
 'I6',
 'I7',
 'I8',
 'Vtrsin',
 'Mono',
 'Fentes',
 'Moteurs',
 'Moteurs2',
 'Compteurs',
 'Compteurs2']
```

Mise à jour des Valeurs

Les conversions absolu/relatif se basent sur la valeur courante stockée dans la grandeur et ne font pas de mise à jour automatique - pour assurer cette mise à jour vous pouvez appeler la méthode `ensureuptodate` sur la grandeur avant de réaliser vos conversions. Lorsque vous appelez des méthodes `set`, `get`... PyExp s'assure que les mises à jour ont été effectuées avant de faire les calculs absolu/relatif nécessaires.

Méthodes Bloquantes

Les méthodes et fonctions de PyExp sont synchrones, c'est à dire qu'elles ne retournent que lorsqu'elles ont terminé ce qu'elles doivent réaliser et bloquent donc la suite de l'exécution du script.

Ceci permet d'avoir une vision séquentielle des opérations. Si toutefois vous avez besoin d'effectuer une opération de façon asynchrone, vous pouvez soit utiliser n'importe quelle fonction avec un objet Asynch, soit utiliser les méthodes normales des grandeurs (et des autres acteurs) en les postfixant par **_a** ; ces méthodes retournent alors un objet Asynch et c'est dans ce cas à vous de gérer la synchronisation dans le code de vos scripts (voir outils du module asynchmod).

Chapitre 3. Configuration

La configuration de PyExp utilise l'outil confmod, et se base sur la variable d'environnement SIE_PYEXP_CONFIG afin d'accéder à la source des informations de configuration (typiquement cette variable d'environnement contient `fileconf:` suivi du chemin d'accès complet au fichier de configuration). Cette source de configuration contient des **entités** de configuration qui décrivent chacune un acteur de pyexp (grandeur, device ou équipement). Pour chaque entité de configuration, les informations sont organisées par section et identifiées par des noms.

Nous n'allons pas entrer ici dans tous les détails des valeurs possibles pour les configurations - pour cela vous pouvez consulter les descriptions des classes dans le document *PyExp - Machinerie*. Nous donnerons simplement des exemples commentés de configurations pour les différents acteurs que vous pouvez avoir à créer (exemples au format `fileconf`), avec des renvois vers les sections plus précises du document *PyExp - Machinerie*. A terme, un outil graphique devrait permettre d'éditer ce fichier de configuration avec une aide en ligne directe.

Généralités

Format des exemples

Les exemples seront donnés dans le format des fichiers du protocole de configuration `fileconf`, très proche des fichiers configuration au format texte :

```
# Exemple de fichier de configuration.❶
@MATHS❷
  [constantes]❸
    pi=3.14❹
    e=2.71
  [noms]
    geom=["Euclide","Euler","Gauss"]
    arithm=["Fermat","Wiles"]
```

❶ Juste un commentaire dans le fichier de configuration (ligne commençant par #).

❷ Définit le début des informations concernant une entité de configuration nommée "`MATHS`".

❸ Définit le début d'une section de configuration pour l'entité courante, section nommée "`constantes`".

❹ Définit une valeur de configuration 3.14 nommée "`pi`". Les valeurs peuvent être toute expression Python évaluable (entiers, flottants, chaînes de caractères, tuples, listes, dictionnaires…).

Noms des entités de configuration

Afin de pouvoir faire facilement le lien entre un acteur PyExp et l'entité de configuration correspondante, et afin d'identifier rapidement la catégorie de cet acteur à partir de la configuration, les noms des entités de configuration des acteurs PyExp sont composés d'un préfixe suivi d'une barre oblique puis du nom de l'acteur. Les préfixes utilisés sont `PHYSICAL` pour les acteurs grandeur physique, `GROUP`

pour les acteurs de gestion d'équipements, et `DEVICE` pour les acteurs de pilotage de matériels. Exemples :

PHYSICAL/E	GROUP/Mono	DEVICE/MotMM4006_2
PHYSICAL/Time	GROUP/Diffracto	DEVICE/EchOrtec4
PHYSICAL/Theta	GROUP/SlitSet3	

Liens entre acteurs

Quel que soit le type de l'acteur, la liste des upslots et des downslots ainsi que la liste des liens vers d'autres acteurs (à partir des downslots) est toujours présentée de la même façon dans la configuration (les exemples se réfèrent aux acteurs décrits dans la section intitulée *Liens entre acteurs* dans Chapitre 1).

Tout d'abord la section ACTOR contient des noms de valeurs `upslots` et `downslots` avec pour chacun la liste des noms de slots. Typiquement pour une grandeur physique on a les noms de slots suivants :

```
[ACTOR]
  upslots=["physical"]
  downslots=["performer"]
```

Et pour le groupe Mono :

```
[ACTOR]
  upslots=["energy","lambda"]
  downslots=["bragg","trans"]
```

Ensuite, pour chaque acteur une section DOWNLINKS donne la liste des liens issus de ses downslots vers les upslots d'autres acteurs. Les noms des downslots servent de noms de valeurs dans la section, et les valeurs associées indiquent les noms de l'acteur et du upslot pointés par le lien. Par exemple pour la grandeur E :

```
[DOWNLINKS]
  performer=("Mono","energy")
```

Et pour le groupe Mono :

```
[DOWNLINKS]
  bragg=("Bragg","physical")
  trans=("Tr","physical")
```

Par convention on évitera de créer des liens directs entre des acteurs de type groupe et/ou device. On passera par des acteurs grandeurs physiques intermédiaires.

Configuration d'une Grandeur simple

Les grandeurs "simples" sont les grandeurs qui n'offrent que les méthodes de base (get, set, measure…). Elles peuvent être liées à n'importe quel groupe ou device afin de présenter les services fournis par ceux-ci.

Identification de la grandeur

Une configuration typique commence avec :

```
@PHYSICAL/Zozo❶
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement❷
    upslots=["physical"]❸
    downslots=["performer"]❹
    description="Controle du Zozo de l'experience."❺
  [DOWNLINKS]
    performer=("acteurX","slotN")❻
```

❶ Où `Zozo` doit être le nom de la grandeur physique (tel qu'il sera appelé dans le getactor).

❷ La classe de base des acteurs grandeurs physiques : `PhysElement`.

❸ Les grandeurs physiques doivent comporter un upslot `physical` par lequel arrivent les requêtes concernant la grandeur.

❹ Les grandeurs physiques doivent comporter un downslot `performer` par lequel peuvent être transmises des requêtes vers d'autres acteurs.

❺ La description de la grandeur (purement informationel).

❻ Le lien issu du downslot `performer`.

Paramétrage de la conversion

Comme cela a déjà été présenté rapidement dans la section intitulée *Unités et conversions* dans Chapitre 1, à toute grandeur sont associés deux unités et un système permettant les conversions entre ces unités. Une section de configuration est dédiée au paramétrage de cette conversion :

```
  [CONVERTER]
    userdatatype=float❶
    datadatatype=int❷
    convclass=linear❸
    coefh2l=10.0❹
    ofsh2l=0.0❺
```

❶ Le manipulateur pour le type des données au niveau user unit.

❷ Le manipulateur pour le type des données au niveau data unit.

❸ Le convertisseur de données qui sera utilisé. Les paramètres coefficient (`coefh2l`) et offset (`ofsh2l`) sont liés à ce convertisseur. Si le paramètre `convclass` n'est pas rempli, le convertisseur `linear` est utilisé par défaut.

❹ Le coefficient de conversion linéaire.

❺ L'offset de conversion linéaire.

Manipulateurs de données

 **Intérêt**

> Ces manipulateurs permettent à PyExp de gérer les valeurs des grandeurs physiques indépendamment des types de ces valeurs. Les grandeurs physiques peuvent ainsi être associées à des données simples (entiers, flottants, chaînes de caractères), à des tableaux à une dimension (vecteurs), à des tableaux à deux dimensions (images), ou à tout type de données pour lequel un manipulateur est créé. Toutes les opérations de base réalisées sur ces données (sauvegarde, copie en mémoire, comparaison…) sont déléguées aux manipulateurs.

La spécification de manipulateurs de données pour l'unité user et pour l'unité data est obligatoire. La spécification du manipulateur est réalisée en indiquant une classe liée au type de donneés, soit en donnant le nom complet de cette classe (par exemple `pyexp.DataManipIntMod.DataManipInt`), soit pour les classes internes de PyExp en donnant un alias (par exemple `int`), soit pour les autres classes en donnant directement le nom de la classe (à la condition que le module reprenne le nom de la classe suivi de `Mod`, et qu'il soit directement accessible dans le PYTHON_PATH) . Les différentes classes de manipulateurs actuellement définies (et leurs alias) sont listées dans le tableau suivant (d'autres manipulateurs de données pourront être ajoutés ultérieurement, voir les modules `DataManipXXXMod.py` de PyExp.).

Tableau 3-1. Manipulateurs pour les types de données

Classe PyExp	Alias	Manipule
DataManipInt	`int` ou `long`	Nombre entier.
DataManipFloat	`float`	Nombre flottant.
DataManipInt1D	`[int]` ou `[long]`	Tableaux à une dimension de nombres entiers.
DataManipFloat1D	`[float]`	Tableaux à une dimension de nombres flottants.
DataManipInt2D	`[[int]]` ou `[[long]]`	Tableaux à deux dimensions de nombres entiers.
DataManipFloat2D	`[[float]]`	Tableaux à deux dimensions de nombres flottants.
DataManipStr	`str`	Chaîne de caractères.
DataManipStr1D	`[str]`	Tableau à une dimension de chaînes de caractères.
DataManipStr2D	`[[str]]`	Tableau à deux dimensions de chaînes de caractères.
DataManipArray	`numeric` ou `array`	Données du module d'extension Numeric (NumPy).

Si vous n'utilisez pas l'alias, vous devez spécifier le chemin complet d'accès à la classe en incluant le nom du module, et éventuellement le nom du package (par exemple pour la classes `DataManipInt` de PyExp : `pyexp.DataManipIntMod.DataManipInt`).

Convertisseur de données

 **Intérêt**

> La séparation entre la grandeur et le convertisseur de données unité user versus unité data (ainsi que l'utilisation des manipulateurs de données) permet d'avoir une classe grandeur physique générique offrant une série de comportements standards, et de réutiliser cette classe dans différents contextes simplement en fournissant les convertisseurs et manipulateurs adéquats.

La spécification de convertisseur de données est optionnelle, par défaut un convertisseur `linear` est utilisé. La spécification du convertisseur est réalisée en indiquant une classe de conversion, soit en donnant le nom complet de cette classe (par exemple `pyexp.DataConvLinearMod.DataConvLinear`), soit pour les classes internes de PyExp en donnant un alias (par exemple `linear`), soit pour les autres classes en donnant directement le nom de la classe (à la condition que le module reprenne le nom de la classe suivi de `Mod`, et qu'il soit directement accessible dans le PYTHON_PATH). Les différentes classes de convertisseurs actuellement définies (et leurs alias) sont listées dans le tableau suivant (d'autres convertisseurs de données pourront être ajoutés ultérieurement, voir les modules `DataConvXXXMod.py` de PyExp.).

Tableau 3-2. Convertisseurs pour les données

Classe PyExp	Alias	Convertit
DataConvLinear	`linear`	En effectuant une conversion de la forme y=ax+b.
DataConvNone	`none`	N'effectue aucune conversion.

Si vous n'utilisez pas l'alias, vous devez spécifier le chemin complet d'accès à la classe en incluant le nom du module, et éventuellement le nom du package (par exemple pour la classes `DataConvLinear` de PyExp : `pyexp.DataConvLinearMod.DataConvLinear`).

Convertisseur "linear". Les convertisseurs "linéaires" permettent d'effectuer une simple transformation affine `y=ax+b`, ou plus explicitement : `data = coefh2l * user + ofsh2l`[1]. La classe de conversion linéaire est `DataConvLinear` (alias `linear`), c'est la classe de conversion utilisée par défaut pour une grandeur si aucune classe n'est indiquée. Le coefficient `coefh2l` de conversion d'unité user en unité data prend par défaut la valeur `1.0`. L'offset `ofsh2l` de conversion d'unité user en unité data prend par défaut la valeur `0.0`. Il est possible d'utiliser cette classe avec d'autre types de données pour le coefficient et l'offset - pour autant que les valeurs converties supportent la multiplication/division et l'addition/soustraction avec ces données.

Autres paramètres

Divers paramètres permettent de caractériser le fonctionnement de la grandeur :

```
[PHYSICAL]
  lowdlimit=-120❶
  highdlimit=400❷
  unit="mm"❸
  unitverb="millimetres"❹
  dataunit="1/10mm"❺
  dataunitverb="dixiemes de millimetres"❻
  saveformat={"ftxtformat":"%0.3f"}❼
  dsaveformat={"itxtformat":"%-0.4d"}❽
```

❶ Limite basse exprimée en unité data.

❷ Limite haute exprimée en unité data.

❸ Nom cours de l'unité user.

❹ Nom long de l'unité user.

❺ Nom court de l'unité data.

❻ Nom long de l'unité data.

❼ Format de sauvegarde en unité user.

❽ Format de sauvegarde en unité data.

Limites. Les limites haute et basse sont exprimées en unité data, elles sont optionnelles. Lorsque la valeur associée à la grandeur doit être fixée, la nouvelle valeur donnée est vérifiée par rapport aux limites (une borne manquante correspond à une absence de limite).

Transfert de compétences. Une valeur de configuration `[PHYSICAL]forwardlimits` permet d'indiquer à la grandeur qu'elle doit transmettre les requêtes concernant les limites de sa valeur vers l'acteur connecté à son slot performer. Dans notre exemple précédent avec une grandeur E liée à un groupe Mono, lui-même lié à une grandeur Bragg, il est mieux de laisser le monochromateur gérer les calculs de limites en énergie à partir les limites angulaires de la grandeur Bragg et du cristal/réseau sélectionné dans le monochromateur.

Formats de sauvegarde. Dans ses fonctions de sauvegarde, PyExp utilise des dictionnaires d'options. Les paramètres `saveformat` et `dsaveformat` des grandeurs physiques permettent de spécifier via des dictionnaires d'options comment les données (en unité utilisateur et en unité data) doivent être formatées. Les noms des options dépendant des types des données, par exemples :

clé `itxtformat` pour des nombres entiers écrits en format texte (défaut `"%d"`)
clé `ftxtformat` pour des nombres flottants écrits en format texte (défaut `"%g"`)
clé `colsep` pour le séparateur de colonnes dans les tableaux (défaut `"\t"`)
…

Pour les détails des options de formatage et de leurs valeurs par défaut, voir .

Sauvegardes au format binaire

Actuellement, seules les sauvegardes au format texte sont réellement implémentées. Les corps des méthodes `tool_writebinstream…` de la classe `DataConverter` sont vides (lèvent une exception), et restent à coder.

Paramètre contrôlé

Par défaut le paramètre contrôlé par les méthodes et fonctions set/get et transmis par le slot performer est le paramètre `VALUE`. Il est possible de positionner une valeur de configuration `[PHYSICAL]performervalname` afin d'indiquer que l'on désire contrôler un autre paramètre. Ceci permet par exemple de créer une grandeur physique `Vx` dont le slot performer est lié au slot `physical` d'une grandeur `X`, et de paramétrer `Vx` afin qu'il contrôle le paramètre `SPEED` de `X`. Agir sur la valeur de `Vx` correspondrait à manipuler la vitesse de `X`.

Configuration d'un groupe de fentes

Identification du groupe de fentes

Une configuration typique commence avec :

```
@GROUP/Fentes❶
  [ACTOR]
    actclass=pyexp.pyexpgroups.SlitSetMod.SlitSet❷
    upslots=["gap1","ofs1","gap2","ofs2"]❸
    downslots=["topleft1","bottomright1","topleft2","bottomright2"]❹
    description="Gestion de toutes les fentes de l'experience."❺
  [DOWNLINKS]
    topleft1=("Levre1G","physical")❻
    bottomright1=("Levre1D","physical")❼
    topleft2=("Levre2H","physical")
    bottomright2=("Levre2B","physical")
```

❶ Le nom donné à l'acteur fentes, principalement pour la gestion des liens (l'utilisateur final ne manipule normalement jamais directement cet acteur).

❷ La classe des groupes fentes : `SlitSet`.

❸ Les upslots permettant de piloter les ouvertures et les centrages des fentes.

❹ Les downslots permettant de transmettre les requêtes de mouvement sur les lèvres des fentes.

❺ La description du groupe de fentes (purement informationel).

❻ Le lien issu du downslot `topleft1` vers la grandeur chargée de contrôler une partie de la géométrie de la fente.

❼ Le lien issu du downslot `bottomright1` vers la grandeur chargée de contrôler une autre partie de la géométrie de la fente.

Slots. Les fentes sont numérotées de 1 à N. Pour chaque numéro de fente il doit y avoir un upslot `gap` et un upslot `ofs` portant le même numéro que la fente. Et pour chaque numéro de fente il doit aussi y avoir un downslot `topleft` et un downslot `bottomright` portant le même numéro que la fente.

Paramétrage d'une fente

Une section de configuration pour une fente du groupe contient :

```
  [SLIT2]❶
    direction="verticale"❷
    geometry=STANDARD❸
```

❶ Pour chaque fente gérée par le groupe, une section particulière permet de spécifier ses caractéristiques (les fentes étant numérotées de 1 à N, les sections sont nommées `SLIT1` à `SLITN`).

❷ Indication de direction de la fente (purement informationel).

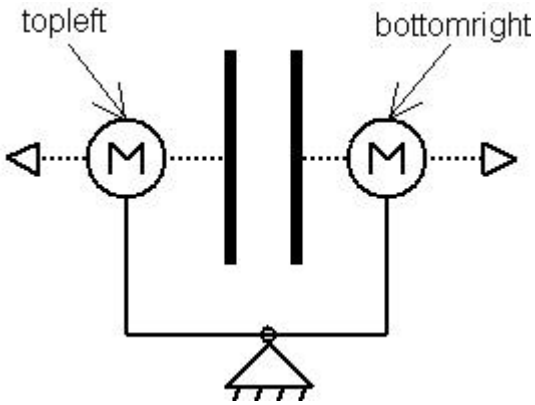
❸ Géométrie de la fente pour que le groupe sache quels calculs effectuer pour piloter les moteurs. Les différentes valeurs possibles sont : STANDARD, LEFTRELATIVETORIGHT, RIGHTRELATIVETOLEFT, INDEPENDANT. Voir la description de ces géométries ci-dessous.

Géométries

Le groupe permet de s'adapter à divers systèmes mécaniques de fentes. Suivant la géométrie, il répartit différement les mouvements à effectuer entre les deux grandeurs liées aux deux downslots de contrôle de la fente. Normalement ces grandeurs sont liées à des contrôles d'axes. Il faut paramétrer le système de conversion de ces deux grandeurs afin qu'elles travaillent dans la même unité, et que leur sens de déplacement croissant corresponde à ce qui est décrit dans la géométrie. Il est conseillé de régler le zéro de ces deux grandeurs afin que cela corresponde à une position fermée et centrée de la fente (pour avoir un zéro fermé et centré, il est aussi possible de faire des ajustements au niveau des systèmes de conversion des grandeurs liées aux upslots `gap` et `ofs`).

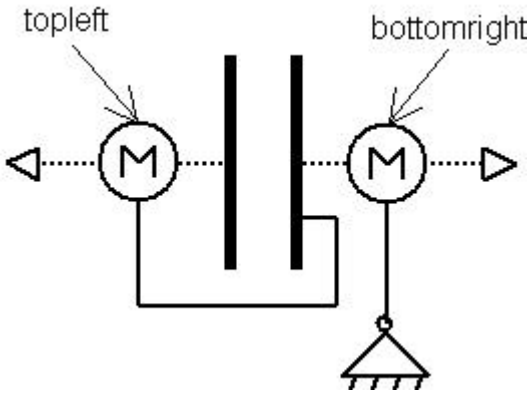
STANDARD. Les deux lèvres sont pilotées indépendament, chacune par son moteur. La lèvre gauche/haute est pilotée via la grandeur liée au slot `topleft`. La lèvre droite/basse est pilotée via la grandeur liée au slot `bottomright`.

Figure 3-1. Géométrie de fente STANDARD



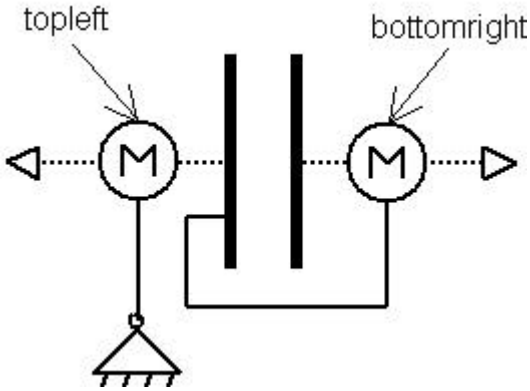
LEFTRELATIVETORIGHT. La lèvre gauche/haute est liée au mouvement de la lèvre droite/bas. L'écart entre la lèvre gauche/haute et la lèvre droite/bas est piloté via la grandeur liée au slot `topleft`, en agissant sur la position de la lèvre gauche/haut par rapport à la position de la lèvre droite/bas. Le support des deux lèvres est piloté via la grandeur liée au slot `bottomright`.

Figure 3-2. Géométrie de fente LEFTRELATIVETORIGHT



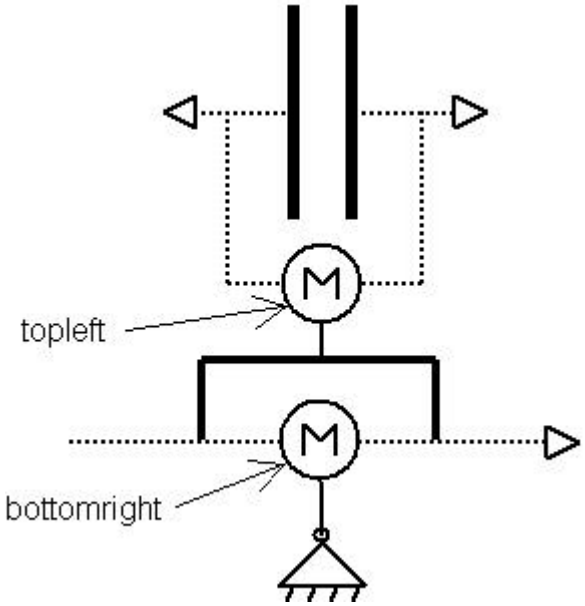
RIGHTRELATIVETOLEFT. La lèvre droite/bas est liée au mouvement de la lèvre gauche/haute. L'écart entre la lèvre droite/bas et la lèvre gauche/haute est pilotée via la grandeur liée au slot `bottomright`, en agissant sur la position de la lèvre droite/bas par rapport à la position de la lèvre gauche/haut. Le support des deux lèvres est piloté via la grandeur liée au slot `topleft`.

Figure 3-3. Géométrie de fente RIGHTRELATIVETOLEFT



INDEPENDANT. L'ouverture et le centrage sont chacun pilotés par des moteurs indépendants. L'ouverture est pilotée via la grandeur liée au slot `topleft`. Le centrage est piloté via la grandeur liée au slot `bottomright`.

Figure 3-4. Géométrie de fente INDEPENDANT



Unités

L'unité présentée pour une fente par ses upslots `gap` et `ofs` est la même que celle fournie par les grandeurs physiques connectées aux downslots `topleft` et `bottomright` de cette fente. Si vous désirez modifier cette unité au niveau utilisateur, il est possible de le faire via le système de conversion des grandeurs liées aux upslots `gap` et `ofs`.

Notes

1. Dans le convertisseur lui-même on parle de `low` pour `data` et de `high` pour `user` - ceci est à rapprocher de la représentation schématique d'une grandeur, qui reçoit des requêtes par le haut via son upslot `physical` avec des valeurs exprimées en unité user, et qui transmet des requêtes par le bas via son downslot `performer` avec des valeurs exprimées en unité data.

Annexe A. Configuration

Fichier de configuration du premier exemple d'utilisation de PyExp. Tous les devices sont des device de simulation permettant de faire fonctionner des scripts sans interagir avec du matériel (les sections `SIMUL` stockent des valeurs pour ces simulations).

```
@PHYSICAL/E
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Energie de l'experience."
  [DOWNLINKS]
    performer=("Mono","energy")
  [PHYSICAL]
    unit="eV"
    unitverb="electron-volts"
    dataunit="eV"
    dataunitverb="electron-volts"
    forwardlimits=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/TRSIN
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Axe de translation du bras sinus du mono."
  [DOWNLINKS]
    performer=("Moteurs","axis4")
  [PHYSICAL]
    unit="cm"
    unitverb="centimetres"
    dataunit="mm"
    dataunitverb="millimetres"
    lowdlimit=0
    highdlimit=15000
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=int
    coefh2l=10.0
    ofsh2l=100.0

@PHYSICAL/Levre1G
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre gauche fente 1"
  [DOWNLINKS]
    performer=("Moteurs","axis5")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/Levre1D
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre droite fente 1"
  [DOWNLINKS]
    performer=("Moteurs","axis6")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/Levre2H
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre haut fente 2"
  [DOWNLINKS]
    performer=("Moteurs","axis7")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
```

```
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/Levre2B
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre bas fente 2"
  [DOWNLINKS]
    performer=("Moteurs","axis8")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/Levre3G
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre gauche fente 3"
  [DOWNLINKS]
    performer=("Moteurs2","axis1")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/Levre3D
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre droite fente 3"
  [DOWNLINKS]
    performer=("Moteurs2","axis2")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/Levre4H
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre haut fente 4"
  [DOWNLINKS]
    performer=("Moteurs2","axis3")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/Levre4B
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Moteur levre bas fente 4"
  [DOWNLINKS]
    performer=("Moteurs2","axis4")
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F1gap
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Gap fente 1"
  [DOWNLINKS]
    performer=("Fentes","gap1")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F1ofs
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Offset fente 1"
  [DOWNLINKS]
```

```
    performer=("Fentes","ofs1")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F2gap
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Gap fente 2"
  [DOWNLINKS]
    performer=("Fentes","gap2")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F2ofs
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Offset fente 2"
  [DOWNLINKS]
    performer=("Fentes","ofs2")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F3gap
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Gap fente 3"
  [DOWNLINKS]
    performer=("Fentes","gap3")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F3ofs
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Offset fente 3"
  [DOWNLINKS]
    performer=("Fentes","ofs3")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F4gap
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Gap fente 4"
  [DOWNLINKS]
    performer=("Fentes","gap4")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/F4ofs
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Offset fente 4"
  [DOWNLINKS]
```

```
    performer=("Fentes","ofs4")
  [PHYSICAL]
    forceabsolutedset=1
  [CONVERTER]
    convclass=linear
    userdatatype=float
    devdatatype=float
    coefh2l=1.0
    ofsh2l=0.0

@PHYSICAL/I1
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 1"
  [DOWNLINKS]
    performer=("Compteurs","counter1")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/I2
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 2"
  [DOWNLINKS]
    performer=("Compteurs","counter2")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/I3
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 3"
  [DOWNLINKS]
    performer=("Compteurs","counter3")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/I4
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 4"
  [DOWNLINKS]
    performer=("Compteurs","counter4")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/I5
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 5"
  [DOWNLINKS]
    performer=("Compteurs2","counter1")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/I6
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 6"
  [DOWNLINKS]
    performer=("Compteurs2","counter2")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/I7
  [ACTOR]
```

```
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 7"
  [DOWNLINKS]
    performer=("Compteurs2","counter3")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/I8
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Intensite detecteur 8"
  [DOWNLINKS]
    performer=("Compteurs2","counter4")
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1
    ofsh2l=0

@PHYSICAL/Vtrsin
  [ACTOR]
    actclass=pyexp.PhysElementMod.PhysElement
    upslots=["physical"]
    downslots=["performer"]
    description="Vitesse du moteur pilotant le bras sinus."
  [DOWNLINKS]
    performer=("trsin","physical")
  [PHYSICAL]
    unit="step/sec"
    unitverb="steps per second"
    dataunit="step/sec"
    datasunitverb="steps per second"
    lowdlimit=10
    highdlimit=5000
    forceabsolutedset=1
    performervalname="SPEED"
  [CONVERTER]
    convclass=linear
    userdatatype=int
    devdatatype=int
    coefh2l=1.0
    ofsh2l=0.0

@GROUP/Mono
  [ACTOR]
    actclass=pyexp.pyexpgroups.MonoSU3Mod.MonoSU3
    upslots=["energy","lambda"]
    downslots=["trans"]
    description="Monochromateur de l'experience."
  [DOWNLINKS]
    trans=("Trsin","physical")
  [GRATE]
    current=R1
    R1=(1800,"eV",70,300,"high energy",{"incidence":84.0,"coef":2074.66,"zeroorder":0.0 })
    R2=(900,"eV",22,155,"medium energy",{"incidence":80.0,"coef":923.67,"zeroorder":0.0})
    R3=(300,"eV",4,40,"low energy",{"incidence":77.0,"coef":237.53,"zeroorder":0.0})
  [SU3MONO]
    sh_length=290.35
    outofgear_pos=-70.000

@GROUP/Fentes
  [ACTOR]
    actclass=pyexp.pyexpgroups.SlitSetMod.SlitSet
    upslots=["gap1","ofs1","gap2","ofs2","gap3","ofs3","gap4","ofs4"]
    downslots=["topleft1","bottomright1","topleft2","bottomright2","topleft3","bottomright3","topleft4","bottomright4"]
  [DOWNLINKS]
    topleft1=("Levre1G","physical")
    bottomright1=("Levre1D","physical")
    topleft2=("Levre2H","physical")
    bottomright2=("Levre2B","physical")
    topleft3=("Levre3G","physical")
    bottomright3=("Levre3D","physical")
    topleft4=("Levre4H","physical")
    bottomright4=("Levre4B","physical")
  [SLIT1]
    direction="verticale"
    geometry=STANDARD
  [SLIT2]
    direction="horizontale"
    geometry=LEFTRELATIVETORIGHT
  [SLIT3]
    direction="verticale"
    geometry=RIGHTRELATIVETOLEFT
  [SLIT4]
    direction="horizontale"
    geometry=INDEPENDANT

@DEVICE/Moteurs
  [ACTOR]
    actclass=pyexp.pyexpdevs.DevSimulMod.DevSimul
    upslots=["axis1","axis2","axis3","axis4","axis5","axis6","axis7","axis8"]
    downslots=[]
  [SIMUL]
```

```
    axis2,VALUE=0
    axis4,VALUE=0
    axis7,VALUE=10.0
    axis1,VALUE=0
    axis5,VALUE=5.0
    axis6,VALUE=5.0
    axis3,VALUE=0
    axis8,VALUE=105.0
    axis8,SIMULSETVALUEDELAY=5

@DEVICE/Moteurs2
  [ACTOR]
    actclass=pyexp.pyexpdevs.DevSimulMod.DevSimul
    upslots=["axis1","axis2","axis3","axis4","axis5","axis6","axis7","axis8"]
    downslots=[]
  [SIMUL]
    axis7,VALUE=0
    axis1,VALUE=5.0
    axis5,VALUE=0
    axis2,VALUE=10.0
    axis4,VALUE=0.0
    axis6,VALUE=0
    axis3,VALUE=10.0
    axis8,VALUE=0

@DEVICE/Compteurs
  [ACTOR]
    actclass=pyexp.pyexpdevs.DevSimulMod.DevSimul
    upslots=["counter1","counter2","counter3","counter4"]
    downslots=[]

@DEVICE/Compteurs2
  [ACTOR]
    actclass=pyexp.pyexpdevs.DevSimulMod.DevSimul
    upslots=["counter1","counter2","counter3","counter4"]
    downslots=[]
```

PyExp - Machinerie

Laurent Pointal
Informaticien, développeur d'applications
C.N.R.S.

PyExp - Machinerie
par Laurent Pointal

PyExp offre, autant à l'utilisateur qui prépare des scripts expérimentaux qu'à celui qui programme la gestion d'un matériel, la possibilité d'écrire des codes rapidement et en se concentrant sur l'essentiel, pour peu qu'ils utilisent les modèles et outils qui sont mis à leur disposition. Il existe tout une série de modules externes : configuration, synchronisations multithread, log… (outils utilisables indépendamment de PyExp). Et surtout un certain nombre de comportements automatiques sont directement hérités des classes parentes du noyau de PyExp.

Le noyau de PyExp se charge principalement de gérer la mise en place de l'infrastructure d'objets associée au poste expériemental, le routage des requêtes entre ces objets, de la gestion du multithread pour l'exécution des requêtes en parallèle sur différents matériels, et des remontées de valeurs.

Table des matières

Liste des tableaux

Liste des illustrations

Chapitre 1. Introduction

Conventions

Lors du développement de pyexp, nous avons adopté certaines conventions d'écriture et de nomage.

Pour les symboles risquant d'être utilisés en ligne de commande, on évite autant que possible l'emploi de caractères avec des casses mixées (minuscules et majuscules en même temps), ainsi que l'emploi du caractère underscore (_). On essaie aussi d'avoir des noms courts afin de faciliter la saisie (mais signifiants - toujours le même problème de compromis)

Pour le code, on utilise des termes anglophones, facilitant l'adoption et l'utilisation du logiciel par des non francophones. De même, les commentaires sont en anglais (par contre la documentation actuelle est en français).

Afin de limiter les pollutions d'espaces de nom lors des `import *`, on utilise la déclaration __all__ au début du script. Elle donne explicitement les noms exportés par le module lors des `import *` (cela n'empêche pas l'import individuel d'un symbole si quelqu'un le veut). Exemple :

```
__all__ = [ "nom1", "nom2", "nom3" ]
```

A noter que l'on n'exporte **jamais** de symbole de variable globale par ce biais (voir pourquoi dans la section sur les variables globales plus loin).

Modules

Les modules qui contiennent (et on été créés pour) une classe particulière sont nommés d'après le nom de cette classe, postfixé par "`Mod`" (classe `DataConvBidon` → `DataConvBidonMod`). Ils ne seront normalement jamais chargés "à la main", mais plutôt dans des scripts automatiques à partir d'informations de configuration.

Les modules qui contiennent des outils, des fonctions ou scripts d'utilisation courante, sont nommés en lettres minuscules. Cela facilitera leur chargement à partir d'une console, et l'écriture des scripts qui accèdent aux membres du module par son nom.

Constantes

Les constantes utilisent des noms tout en majuscules. On peut y utiliser l'underscore (_) afin de faciliter la lecture de certains noms (l'absence d'identification de début de mot pouvant rendre la lecture difficile).

Variables

Les variables utilisent des noms tout en minuscules. On peut y utiliser l'underscore (_) afin de faciliter la lecture de certains noms (l'absence d'identification de début de mot pouvant rendre la lecture difficile), mais cela est à éviter autant que possible - surtout dans le cas de variables susceptibles d'être utilisées en ligne de commande.

Globales

Pour les identifier rapidement, on préfixe les variables globales par un caractère `g`. On ne met jamais une variable globale dans la liste des symboles exportés par un module, ceci afin d'éviter les problèmes d'import de noms de variables globales et de désynchronisation ultérieure entre l'objet référencé par le nom dans le module d'import et l'objet référencé par le nom dans le module d'origine : **il faut toujours passer par le nom de son module pour accéder à une variable globale extérieure**.

Membres

Les variables membres devant être préfixées par *`self`* dans les méthodes, on évite le conventionnel (Microsoft) préfix `m_` utilisé souvent avec le C++.

Classes

Les classes utilisent des noms avec une lettre majuscule au début de chaque mot, et des lettres minuscules pour les suites des mots. Elles ne seront généralement pas utilisées directement mais plutôt dans des scripts automatiques à partir d'informations de configuration.

Méthodes

Les méthodes utilisent des noms tout en minuscules. Les méthodes pouvant être redéfinies dans les sous-classes commencent en général par `do_`..., afin de les identifier comme telles. On évite qu'une méthode `do_xxx` doive être appelée directement en proposant une méthode `xxx` qui se charge d'appeler `do_xxx`, et qui est la méthode **normale** pour un utilisateur de mettre en œuvre la fonctionnalité xxx.

Chapitre 2. Système de conversion

Le système de conversion de pyexp permet aux développeurs de disposer d'objets de conversion utilisables en divers endroits du logiciel et facilement interchangeables. Ce système est basé sur une classe abstraite `DataConverter` qui définit une interface de méthodes de conversion de base entre une unité 'high' et une unité 'low'. Ensuite une série de sous-classes implémentent les différentes méthodes de conversion, de nouvelles sous-classes pouvant être développées en fonction des besoins. Le système de conversion est utilisé dans pyexp :

- dans `PhysElement` pour les conversions entre les unités utilisateur et les unités données des grandeurs physiques,
- dans `AxisElement` pour les conversions entre les unités des codeurs et les unités des moteurs,
- dans le support de base des groupes de grandeurs de la classe `GroupElement`, pour les conversions entre les pseudo grandeurs que manipule normalement l'utilisateur et les grandeurs réelles qui vont effectuer les opérations de plus bas niveau.

Manipulateurs. En plus du système de conversion, on dispose aussi d'un système de **manipulation de valeurs**, qui permet aux convertisseurs et aux scripts de pyexp de manipuler les valeurs d'une façon générique, en disposant d'un outils capable de les dupliquer, les enregistrer dans des fichiers ou les relire… Les manipulateurs sont basés sur une classe abstraite `DataManipulator` qui définit une interface de méthodes permettant de réaliser ces différentes opérations. Ensuite une série de sous-classes implémentent les manipulations de valeurs en fonction des types de données, de nouvelles sous-classes pouvant être développées en fonction des besoins.

Ce système permet de créer des convertisseurs et des manipulateurs de données spécifiques, tout en conservant inchangé le cadre général des scripts de pyexp avec l'utilisation de ces convertisseurs et manipulateurs.

Classes de conversion existantes

Tout développeur peut créer des sous-classes de `DataConverter` en fonction des besoins (voir la section suivante), mais pyexp fournit en standard des convertisseurs pour les cas les plus courants :

`DataConvNone`

N'implémente aucune conversion, les fonctions de conversion se contentant de retourner les valeurs qu'on leur donne. Permet de considérer niveaux de pyexp que l'on dispose dans tous les cas d'un convertisseur, et de ne pas avoir à tester le cas particulier de son absence.

`DataConvLinear`

Effectue des conversions linéaires de la forme `y = a × x + b` (où y est exprimé en unité 'low' et x en unité 'high'). Supporte les paramètres coefficient et offset.

Création d'une classe de conversion

Avant de créer une nouvelle classe de conversion, vérifier s'il n'existe pas dans les classes existantes une classe susceptible de répondre à vos besoins (voir section précédente). Il est rappelé que la classe parente des convertisseurs, `DataConverter`, ne fait pas de supposition a priori sur le type de données manipulées par les fonctions de conversion, et utilise des objets extérieurs de manipulation (ceci n'empêche pas un convertisseur de faire des vérifications sur les données qui lui sont fournies afin de vérifier leur compatibilité avec ses algorithmes).

Voici un exemple d'une classe de conversion `DataConvBidon`. Nous plaçons cette classe de conversion dans un module `DataConvBidonMod` (convention généralement appliquée dans pyexp pour les classes d'objets créés dynamiquement à partir des informations de configuration). Les instances de cette classe ont besoin pour travailler de deux paramètres, un paramètre fixe (stocké dans la variable membre `__paramConf`) et un paramètre réglable (stocké dans la variable membre `__paramUser`) ; le calcul de conversion est complètement… bidon.

```
from pyexp.DataConverterMod import *
…
class DataConvBidon (DataConverter) :❶
    def __init__ (self) :❷
        DataConverter.__init__ (self)
        self.characterize (hash2l=1,hasl2h=1,h2lmodsrc=0,l2hmodsrc=0)
        self.__paramConf = 0.0
        self.__paramUser = 0.0

    def do_loadconfig (self,conf,sect) :❸
        try :
            oldparamConf = self.__paramConf
            oldparamUser = self.__paramUser
            self.do_loadConfig (conf,sect)
            self.__paramConf = conf.getckfloat (sect,"paramConf",20.3)
            self.paramUser (conf.getckfloat (sect,"paramUser",3.14))
        except :
            self.__paramConf = oldparamConf
            self.__paramUser = oldparamUser
            raise

    def do_storeconfig (self,conf,sect) :❹
        DataConverter.do_loadConfig (self,conf,sect)
        conf.set("paramConf",self.__paramConf)
        conf.set("paramUser",self.__paramUser)

    def do_h2lconversion (self, value) :❺
        return value * (self.__paramUser ** 2) + self.__paramConf

    def do_l2hconversion (self, value) :❻
        return (value - self.__paramConf) / (self.__paramUser ** 2)

    def do_infos (self) :❼
        s = DataConverter.do_infos (self)
        s += "\t-Think stage: %f\n"%+self.__paramConf
        s += "\t-paramUser (identity level): %f\n"%self.__paramUser
        return s

    def do_paramsdesc (self) :❽
        l = DataConverter.do_paramsdesc (self)
        l.append (("priv","_DataConvBidon__paramConf","float","Think stage"))
        l.append (("pub","paramUser","float","Identity level"))
        return l
```

```
def paramUser (self,uparam=None) :❾
    if uparam != None :
        self.checkusermodifiable ("paramUser")
        old = self.__paramUser
        self.__paramUser = uparam
        return old
    else :
        return self.__paramUser
```

❶ La classe de conversion que vous créez doit hériter de `DataConverter` (ou d'une de ses sous-classes prévue pour être surchargeable).

❷ La méthode d'initialisation de votre objet de conversion doit commencer par appeler la méthode d'initialisation de la classe parente. Ensuite elle doit appeler la méthode `characterize` afin de spécifier ses caractéristiques propres différentes de celles par défaut, **c'est cette caractérisation qui permet un fonctionnement à la fois optimal et sécurisé du système de conversion**. Enfin elle peut créer et initialiser ses variables membres.

❸ Si votre classe de conversion nécessite de relire des informations de configuration, vous devez redéfinir la méthode `do_loadconfig` héritée de `DataConverter`. Votre méthode doit commencer par appeler la méthode de chargement des paramètres de configuration de la classe parente, puis ensuite charger ses paramètres propres (elle peut utiliser ses fonctions d'accès pour contrôler la validité de ces paramètres). Dans note exemple nous avons ajouté un bloc try/except afin de sauvegarder les anciennes valeurs et de les restaurer en cas d'erreur (afin que les valeurs restent cohérentes entre elles).

❹ Si votre classe de conversion nécessite de stocker des informations de configuration, vous devez redéfinir la méthode `do_storeconfig` héritée de `DataConverter`. Votre méthode doit commencer par appeler la méthode de stockage des paramètres de configuration de la classe parente, puis ensuite sauvegarder ses paramètres propres.

❺ Si vous avez une fonction de conversion de l'unité 'high' vers l'unité 'low', vous devez redéfinir la méthode `do_h2lconversion` en y mettant votre code de calcul. Il ne faudra pas oublier d'indiquer que vous supportez ce sens de conversion en indiquant une valeur 1 au paramètre *`hash2l`* donné à la méthode `characterize` lors de l'initialisation de votre objet.

❻ Si vous avez une fonction de conversion de l'unité 'low' vers l'unité 'high', vous devez redéfinir la méthode `do_l2hconversion` en y mettant votre code de calcul. Il ne faudra pas oublier d'indiquer que vous supportez ce sens de conversion en indiquant une valeur 1 au paramètre *`hasl2h`* donné à la méthode `characterize` lors de l'initialisation de votre objet.

❼ Si votre classe de conversion contient des paramètres internes, vous devez redéfinir la méthode `infos` héritée de `DataConverter`. Votre méthode doit commencer par appeler la méthode d'informations héritée de la classe parente, récupérer la chaine retournée et y concaténer ses lignes d'information (lignes sous la forme "\t-nom: valeur\n"). Enfin votre méthode doit retourner la chaine résultante.

❽ Si votre classe de conversion contient des paramètres internes qui doivent pouvoir être modifiés via une interface utilisateur (graphique ou textuelle), vous devez redéfinir la méthode `paramsdesc` héritée de `DataConverter`. Votre méthode doit commencer par appeler la méthode de description des paramètres héritée de la classe parente, récupérer la liste de tuples retournée et y ajouter ses propres tuples de description de paramètres (voir DataConverter.paramsdesc). Enfin votre méthode doit retourner la liste résultante.

❾ Pour les paramètres de conversion modifiables par l'utilisateur nous demandons qu'une fonction d'accès soit définie. Lorsque cette fonction reçoit une valeur `None`, elle se contente de retourner la valeur courante du paramètre. Lorsqu'elle reçoit une valeur pour le paramètre, elle doit d'abord vérifier que l'utilisateur est autorisé à modifier les paramètres de conversion (en appelant la méthode `checkusermodifiable` à laquelle elle donne comme argument le nom du paramètre devant être modifie), puis sauvegarder l'ancienne valeur, et ensuite contrôler la nouvelle valeur et la stocker en place de l'ancienne ; finalement la fonction d'accès retourne l'ancienne valeur.

Classes de manipulation existantes

Tout développeur peut créer des sous-classes de `DataManipulator` en fonction des besoins, mais pyexp fourni en standard des manipulateurs pour les cas les plus courants :

`DataManipInt`

Supporte les manipulations de valeurs entières (entiers standards ou entiers longs Python).

`DataManipInt1D`

Supporte les manipulations de vecteurs 1D de valeurs entières (entiers standards ou entiers longs Python).

`DataManipInt2D`

Supporte les manipulations de matrices 2D de valeurs entières (entiers standards ou entiers longs Python).

`DataManipFloat`

Supporte les manipulations de valeurs flottantes.

`DataManipFloat1D`

Supporte les manipulations de vecteurs 1D de valeurs flottantes.

`DataManipFloat2D`

Supporte les manipulations de matrices 2D de valeurs flottantes.

`DataManipStr`

Supporte les manipulations de valeurs chaînes.

`DataManipStr1D`

Supporte les manipulations de vecteurs 1D de chaînes.

`DataManipStr2D`

Supporte les manipulations de matrices 2D de chaînes.

Types compatibles

Pour le moment les entiers et les entiers longs sont traités en utilisant les mêmes classes de manipulation, et les chaînes standard et chaînes Unicode sont traitées en utilisant les mêmes

classes de manipulation. Si cela pose problème, des classes spécifiques pour les entiers longs et pour les chaînes Unicode devront être créées.

Création d'une classe de manipulation

Avant de créer une nouvelle classe de manipulation, vérifier s'il n'existe pas dans les classes existantes une classe susceptible de répondre à vos besoins (voir section précédente). Il est rappelé que la classe parente des manipulateurs, `DataManipulator`, offre un certain nombre de fonctions ayant un comportement automatique ainsi que des fonctions outils qui évitent de dupliquer certains codes.

Voici un exemple d'une classe de manipulation `DataManipVect3` qui se chargerait d'un vecteur de données à 3 composantes flottantes (ex. repérage dans l'espace). Nous plaçons cette classe de conversion dans un module `DataManipVect3Mod` (convention généralement appliquée dans pyexp pour les classes d'objets créés dynamiquement à partir des informations de configuration).

```
from pyexp.DataManipulatorMod import *
…
class DataManipVect3 (DataManipulator) :❶
    def __init__ (self) :❷
        DataManipulator.__init__ (self)

    def do_makecopy (self,value) :❸
        return value[0:3]

    def do_checkaccept (self,value) :❹
        return type (value) in [types.TupleType, types.ListType] and \
                len(value)==3

    def do_getdimsizes (self,value) :❺
        return (3,)

    def do_getbasemanipulator (self,value) :❻
        return datamanipulator ("float")

    def do_getbasetype (self,value) :❼
        return types.FloatType

    def do_compare (self,v1,v2) :❽
        return DataManipulator.do_compare (self, self.__vlen(v1), \
                self.__vlen(v2))

    def do_add (self,v1,v2) :❾
        return (v1[0]+v2|0], v1[1]+v2[1], v1[2]+v2[2])

    def do_subst (self,v1,v2) :(10)
        return (v1[0]-v2|0], v1[1]-v2[1], v1[2]-v2[2])

    def do_writebinstream (self,value,fileopt,valopt) :(11)
        self.tool_writebinstream_float1D (value,fileopt,valopt,(3,))

    def do_writestrstream (self,value,fileopt,valopt) :(12)
        self.tool_writestrstream_float1D (value,fileopt,valopt,(3,))

    def do_infos (self) :(13)
        s = DataManipulator.do_infos (self)
        s += "\t-data type: vector of 3 float.\n"
        s += "\t-data size: 8 bytes by value.\n"
        return s
```

```
def __vlen (self,v) :(14)
    return math.sqrt (v[0]*v[0]+v[1]*v[1]+v[2]*v[2])
```

❶ La classe de manipulation que vous créez doit hériter de `DataManipulator` (ou d'une de ses sous-classes prévue pour être surchargeable).

❷ La méthode d'initialisation de votre objet de manipulation doit appeler la méthode d'initialisation de la classe parente. Ensuite elle peut éventuellement initialiser ses variables membres si besoin.

❸ Le comportement par défaut de la méthode `do_makecopy` utilise le module `copy` de Python. Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de dupliquer la valeur plus rapidement, alors la méthode `do_makecopy` doit être redéfinie dans votre classe afin de réaliser la copie. Dans notre exemple nous créons simplement et rapidement une copie contenant les 3 mêmes valeurs.

❹ Dans tous les cas vous devez définir la méthode `do_checkaccept` afin de contrôler qu'une valeur est bien supportée par votre classe. Il est à noter que c'est le seul endroit de votre classe où il y a un contrôle sur la valeur, toutes les autres méthodes assument que la valeur qu'elles reçoivent est bien du type que la classe supporte (ou d'un type compatible - ici liste ou tuple de 3 valeurs flottantes).

❺ Dans tous les cas vous devez définir la méthode `do_getdimsizes` afin de retourner les dimensions de la valeur. Dans notre cas la dimension est fixe - vecteur de 3 données - et nous retournons donc un tuple contenant 1 élément (une dimension) de valeur 3 (trois valeurs dans la dimension).

❻ Le comportement par défaut de la méthode `do_getbasemanipulator` se base sur le type retourné par `getbasetype` et traite les types entier, long et flottant. Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de connaître le manipulateur de base plus rapidement, alors la méthode `do_getbasemanipulator` doit être redéfinie dans votre classe. Dans notre exemple nous nous attendons à des valeurs de base en nombres flottants, nous retournons donc directement un manipulateur pour ce type.

❼ Le comportement par défaut de la méthode `do_getbasetype` se base sur les informations retournées par `getdimsizes` afin d'accéder à une valeur individuelle, puis retourne le type de cette valeur. Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de connaître le type de base plus rapidement, alors la méthode `do_getbasetype` doit être redéfinie dans votre classe. Dans notre exemple nous nous attendons à des valeurs de base en nombres flottants, nous retournons donc directement le type flottant.

❽ Le comportement par défaut de la méthode `do_compare` utilise simplement les opérateurs de comparaison entre les valeurs pour trouver quelle constante de comparaison retourner. Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de connaître la constante de comparaison à retourner plus rapidement, alors la méthode `do_compare` doit être redéfinie dans votre classe. Dans notre exemple nous calculons la longueur des 2 vecteurs et basons notre comparaison sur cette longueur en appelant la méthode `do_compare` héritée.

❾ Le comportement par défaut de la méthode `do_add` utilise simplement l'opérateur `+` entre les valeurs.Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de réaliser la somme plus rapidement, alors la méthode `do_add` doit être redéfinie dans votre classe. Dans notre exemple nous calculons un vecteur résultant basé sur une somme des vecteurs composante par composante.

(10) Le comportement par défaut de la méthode `do_subst` utilise simplement l'opérateur `-` entre les valeurs. Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de réaliser la soustraction plus rapidement, alors la méthode `do_subst` doit être redéfinie dans votre classe. Dans notre exemple nous calculons un vecteur résultant basé sur une soustraction des vecteurs composante par composante.

(11) Le comportement par défaut de la méthode `do_writebinstream` se base sur les informations obtenues par les méthodes `getbasetype` et `getdimsizes` ; il supporte les types entier, entier long, flottant et chaîne, pour les valeurs simples, les vecteurs et les matrices 2D il utilise ensuite les méthodes outils pour réaliser les écritures binaires. Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de réaliser l'écriture binaire plus rapidement, alors la méthode `do_writebinstream` doit être redéfinie dans votre classe. Dans notre exemple nous appelons directement la méthode outil correspondant au type de données manipulé.

(12) Si vous désirez ajouter, aux informations de base générées par DataManipulator.do_infos, des informations liées à vos objets de conversion, vous devez redéfinir la méthode `do_infos` dans votre classe. Votre méthode doit commencer par appeler la méthode `do_infos` héritée et récupérer la chaîne d'information retournée. Ensuite elle peut y concaténer ses informations propres, et enfin retourner la chaîne complète.

(13) Le comportement par défaut de la méthode `do_writestrstream` se base sur les informations obtenues par les méthodes `getbasetype` et `getdimsizes` ; il supporte les types entier, entier long, flottant et chaîne, pour les valeurs simples, les vecteurs et les matrices 2D il utilise ensuite les méthodes outils pour réaliser les écritures textuelles. Si ce comportement n'est pas adapté à la valeur manipulée, ou s'il est possible de réaliser l'écriture textuelle plus rapidement, alors la méthode `do_writestrstream` doit être redéfinie dans votre classe. Dans notre exemple nous appelons directement la méthode outil correspondant au type de données manipulé.

(14) Vous pouvez avoir des méthodes spécifiques pour manipuler les données, en général on les rendra privées en utilisant la notation Python `__xxx`. Ici une méthode qui calcule la longueur du vecteur.

Chapitre 3. Pseudo-grandeurs et Groupes

Chaque grandeur représente virtuellement un élément ayant un sens au niveau de l'expérience pilotée, on trouvera donc par exemple tout naturellement une association entre un moteur sur un poste expérimental, et une grandeur au niveau du logiciel d'acquisition. Mais la notion de grandeur physique a un sens plus large dans pyexp, et à côté des grandeurs intimement liées à des périphériques, on trouve des **pseudo-grandeurs** liées à des éléments éventuellement moins concrets.

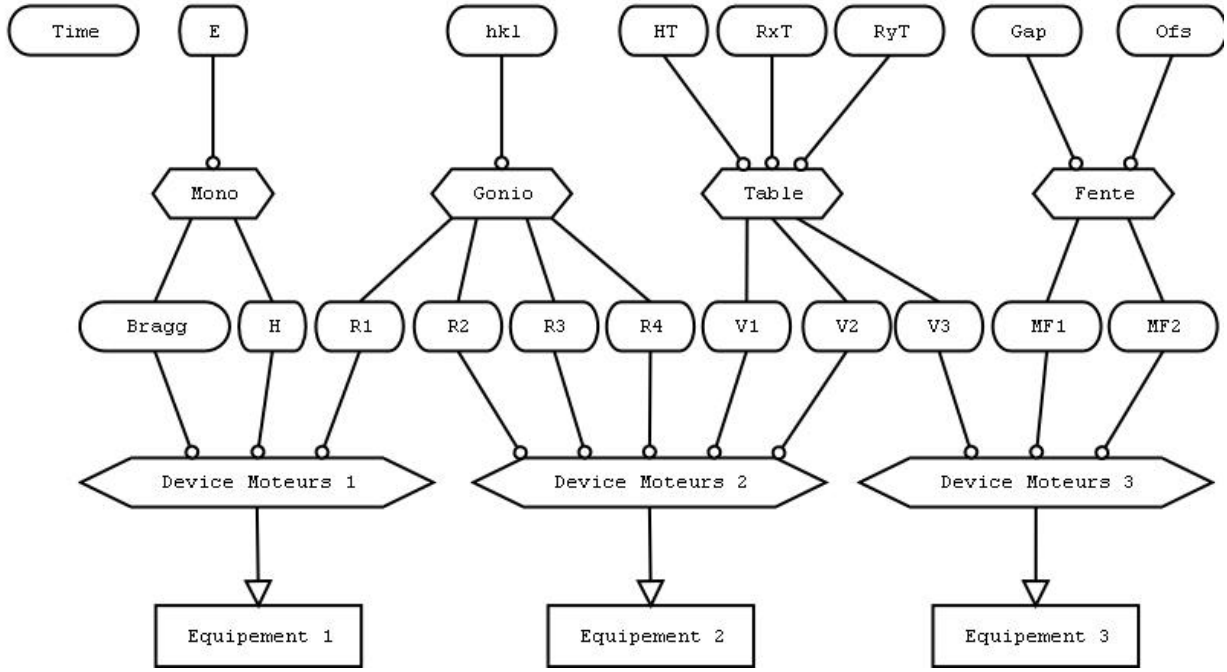
Ces pseudo-grandeurs peuvent être liées à des éléments purement informatiques, par exemple la date ou l'heure sur l'ordinateur, ou encore une information relue dans un fichier…

Elles peuvent aussi être liées à des fonctionnements plus ou moins complexes, mettant en cause d'autres grandeurs suivant des lois spécifiques. On rassemble alors les règles et paramètres régissant ces fonctionnements dans des objets **groupes**. Un groupe est en relation d'une part avec des pseudo-grandeurs qui lui délèguent la réalisation les opérations qui leurs sont demandées (fixation, lecture, mesure…), et d'autre part à des grandeurs (pseudo ou non) sur lesquelles il se repose lui pour réaliser ces opérations.

Voici une série - non exhaustive - d'exemples de pseudo-grandeurs et de groupes.

- Une grandeur `Time`, capable de retourner la date et l'heure tels que fournis par l'ordinateur, et ne nécessitant aucun lien vers un équipement spécifique.
- Une grandeur `E`, représentant l'énergie fixée par un système de monochromatisation, liée à un groupe `Mono`, celui-ci pilotant deux mouvements `Bragg` et `H` pour réaliser cette fonction.
- Une grandeur `hkl`, représentant une position dans l'espace réciproque liant l'échantillon à l'énergie sélectionnée sur le monochromateur, grandeur lié à un groupe `Gonio` pilotant quatre moteurs `R1` à `R4` pour positionner l'échantillon fixé sur le goniomètre.
- Trois grandeurs `HT`, `RxT` et `RyT`, représentant la hauteur et les rotations autour des axes X et Y d'une table, grandeurs liées à un groupe `Table` chargé de contrôler les trois vérins `V1` à `V3` qui supportent physiquement la table.
- Deux grandeurs `Gap` et `Ofs` représentant l'ouverture et la position d'une fente dans un faisceau, liées à un groupe `Fente`, celui-ci pilotant deux mouvements `MF1` et `MF2` pour déplacer les deux lèvres permettant de réaliser ces réglages.

Figure 3-1. Exemple de pseudo-grandeurs, groupes…



Publications et abonnements (todo)

Il est parfois nécessaire à des groupes de s'échanger des informations qui leur permette de régler leur fonctionnement suivant le contexte. Pour cela un système de publication et d'abonnement à des informations nommées a été mis en place dans pyexp. Tout un chacun peut déclarer qu'il publie une valeur sous un nom `"toto"`, qui sera alors accessible par abonnement sous le même nom. Il est à noter que ce système n'est pas exclusivement réservé pour les groupes, mais peut être utilisé à n'importe quel endroit dans pyexp.

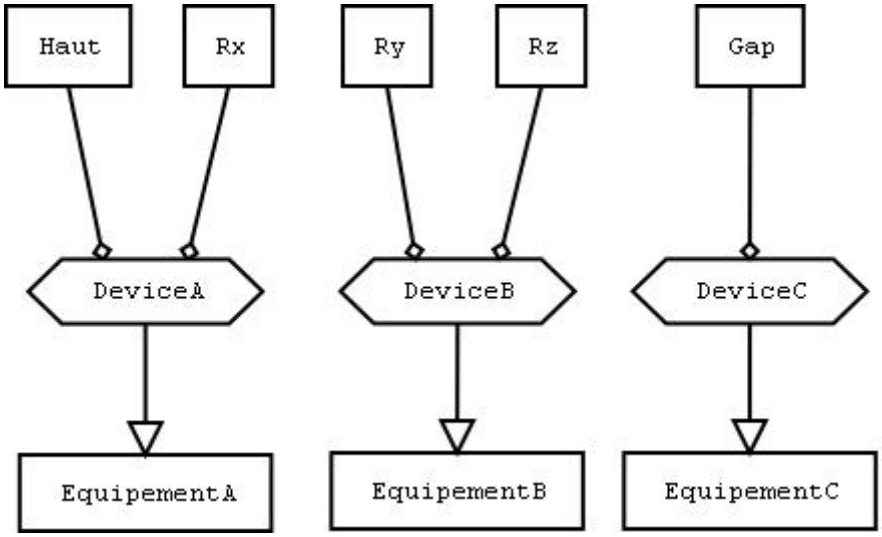
Chapitre 4. Regroupement d'opérations

Un des enjeux de pyexp a été d'offrir aux utilisateurs un interface en terme de grandeurs physiques individuelles, en masquant la façon dont les manipulations de ces grandeurs se traduit effectivement, et de continuer à avoir un interface simple alors que des mécanismes éventuellement complexes sont mis en œuvre afin de réaliser les opérations de façon optimale.

Le problème s'est posé, entre autres, de laisser l'utilisateur exprimer des opérations sur des grandeurs différentes (typiquement des fixations de valeurs ou des lectures), ou encore sur des pseudo-grandeurs dont la manipulation effective nécessite d'intervenir sur plusieurs grandeurs réelles, et de faire réaliser ces opérations de la façon la plus simultanée possible sur les différents équipements concernés.

La solution adoptée consiste à séparer tout ce qui est préparation de la requête (contrôle des paramètres, conversions, reroutage de méthodes…) de leur réalisation effective avec actions sur les matériels. Des méthodes des grandeurs (ainsi que des groupes et des devices) se chargent des préparations et retournent des messages représentant des actions "bas niveau" à effectuer. Chaque message contient entre autres la référence à l'objet destinataire chargé de réaliser effectivement l'action. Les messages sont donc regroupés en listes, par destinataire, et chaque liste est envoyée à son destinataire dans un thread spécifique (ie. les destinataires reçoivent leurs listes d'opérations "en même temps"). A la réception de leurs listes de messages d'actions à réaliser, les destinataires regroupent les requêtes vers le matériel lorsque c'est possible.

Voici l'illustration de ce fonctionnement sur un petit exemple. Prenons une série de grandeurs, `Haut`, `Rx`, `Ry`, `Rz` et `Gap` liés à des périphériques comme indiqué dans le graphique :



Un utilisateur désire fixer simultanément la hauteur, les rotations sur x, y et z, et le gap. Pour cela il appelle la fonction `set` de l'une des façons suivantes

```
set ([Haut,Rx,Ry,Rz,Gap],[12.56,34,65,23.7,0.078])
set (Haut,12.56,Rx,34,Ry,65,Rz,23.7,Gap,0.078)
```

La fonction `set` commence par parcourrir les objets grandeurs liés à

Chapitre 5. Outils pour pyexp

Lors du développement de pyexp, nous avons été amenés à réaliser toute une série d'outils, utiliés dans pyexp mais qui restent autonomes.

Système de log

Nous avons eu besoin dans pyexp d'une fonctionnalité de log souple et portable entre différentes plateformes. Il existe bien un module `syslog`, mais pour les plateformes Unix seulement. Nous avons donc commencé par développer un avatar du module syslog, qui en reprend la même interface, et qui se contente pour le moment d'écrire les logs dans un fichier (il faudra interfacer ce module avec le service/daemon syslogd via les socket dès que possible) - concernant cet avatar, voir la documentation du module `syslog`.

Pour avoir de la souplesse, nous avons défini un module `logger` qui s'occupe lui-même du filtrage des logs, et pemet un plus grande précision dans la sélection des logs que l'on veut voir (fonction de leur origine et de leur importance).

Initialisation

Le module `logger` n'a aucune fonction d'initialisation propre à appeler. Par contre, comme il se base sur le module `syslog` pour la génération des échos vers le système de log… il faut donc penser à initialiser celui-ci en appelant sa fonction `openlog` avec les paramètres désirés. A noter que cette initialisation n'est normalement réalisée qu'une seule fois, par le code d'initialisation du programme, et que cela ne relève normalement pas des modules outils. De son côté, le module `logger` se charge de paramétrer `syslog` avec un appel à la fonction `setlogmask`, de façon à ce que syslog laisse passer tout ce que pourrait générer `logger` (vu que ce dernier s'occupe déjà du filtrage).

Filtrage de logs

Chaque log peut être identifié par un nom de source, qui pemet au module logger de savoir d'où vient le log, et est associé à une catégorie. A partir de ces informations, le module peut appliquer un filtre spécifique suivant la source du log, pour ne laisser passer que les logs utiles pour une situation donnée. Sinon, si la source n'a pas de filtre associé, le module applique un filtre générique. Les filtrages se basent sur la catégorie en utilisant un masque (à chaque catégorie correspond un bit), ce qui permet d'activer certains types de logs indépendamment d'autres, sans notion de niveau de priorité.

Fonctions de génération

Le module `logger` définit une série de fonction permettant de générer des logs suivant leur catégorie. Ces fonctions ont toutes le même format d'appel :

`logxxx` (`<`*`source`*`>`, `<`*`nom-de-fonction`*`>`, `<`*`format`*`>` [,`<`*`arg1`*`>` [,`<`*`arg2`*`>`…]])

Le xxx dépend de la catégorie du log à générer (voir tableau suivant). La *`source`* est un simple chaîne de caractères, s'il n'y en a pas il faut utiliser la valeur `None`. Le *`nom-de-fonction`* permet d'éviter d'avoir à coder cette information dans la chaîne de formatage, les fonctions `logxxx` se chargeant de

l'indiquer automatiquement dans le message de log, s'il n'y en a pas il faut utiliser la valeur `None`. Le format est le message du log, il peut contenir des formatages % auquels devront correspondre des paramètres ***`argi`*** correspondant. S'il y a au moins un paramètre ***`argi`***, alors l'opérateur % est appliqué à la chaîne ***`format`***, sinon la chaîne format est laissée telle quelle (et donc les caractères % ne sont pas interprétés).

Tableau 5-1. Fonctions de log

Fonction	Catégorie de log
`logcritical`	Erreur importante pouvant entraîner la perte de données ou l'arrêt de l'expérience (par exemple un disque de données plein, un périphérique essentiel qui ne répond plus).
`logerror`	Erreur notable, éventuellement suffisante pour stopper un traitement en cours et remonter par une exception.
`logwarning`	Indication d'un problème, l'exécution a continuée (problème annecdotique, ou action de correction). Par exemple, lorsque l'on intercepte une exception et qu'on la corrige, on génère un log de warning.
`logconfig`	Données de configuration, d'où elles viennent, leurs valeurs, si elles sont modifiées…
`logvalues`	Modification de valeurs importantes. Typiquement, les modifications de valeurs dans les grandeurs physique de pyexp.
`logio`	Trace de communication avec un périphérique style RS232, GPIB, réseau. Inclus les appels aux fonctions d'établissement de la communication, d'attente, et tout ce qui est données écrites ou lues sur le périphérique.
`logdevlib`	Trace des appels à des librairies de gestion de périphériques (hors librairies de communication qui sont en logio).
`logdebug`	Informations de débogage importantes.
`logdebuginout`	Trace des entrées et sorties des fonctions et méthodes, des interceptions d'exceptions, de leur retransmission.
`logdebugdump`	Informations de débogage de bas niveau, affichage de données, signalisation de passage…

Format du log

Comme nous nous basons sur `syslog` pour l'envoi du log vers un système plus général, une partie d'entête est automatiquement généré avec une datation à la seconde, le nom de la machine hôte, ainsi que le nom du programme ayant généré le log. Ensuite, le module `logger` ajoute une datation plus précise (milliseconde), la catégorie du log, le nom du thread, le nom de la source si elle a été précisée, le nom de la fonction si elle a été précisée, et enfin le message formaté.

```
>>> logerror ("scanscript","set","Echec a la fixation des valeurs.")

2001-12-10 10:52:58 crocus theApplication: (58439ms) ERRROR      <MainThread> [scanscript]
<set>: Echec a la fixation des valeurs.
```

La datation à la milliseconde s'affiche sur 5 chiffres de 00000 à 59999 ; elle permet de suivre plus précisément les moments où se sont passées certaines actions, ce qui peut être important lors de problèmes de communication avec des matériels. La datation issue du vrai module syslog sous Unix est plutôt du genre `Dec 12 10:52:58`.

Fonctions outils

A côté des fonctions de génération de log, le module `logger` définit une série de fonctions outils : **`enablelog`** et **`disablelog`** permettent d'activer et de désactiver la génération de logs d'une certain catégorie (éventuellement en le spécifiant pour une certaine source), **`registerlogsource`** permet d'indiquer l'existence d'une source de logs (qui pourra avoir son filtre par catégorie propre), **`registeredlogsources`** permet de lister les sources de logs enregistrées, et **`logcategories`** permet de retourner le filtre de catégories courant (éventuellement en spécifiant pour une certaine source), **`catchmsg`** pour récupérer une chaîne simple décrivant l'exception en cours, et **`catchtraceback`** pour récupérer une représentation de la pile d'appel pour l'exception en cours (ces deux fonctions permettent de récupérer des chaînes à logger).

Le module définit aussi deux fonction internes, mais qui peuvent être utiles ailleurs et sont donc exportées : **`thetime`** qui retourne une datation absolue et précise, et **`thetimestr`** qui permet de formater cette datation dans un format standard "`YYYY-MM-JJ hh:mm:ss:uuuu`".

Constantes

Afin de pouvoir indiquer les catégories de log, le module `logger` définit une série de constantes (correspondant aux fonctions homonymes).

Tableau 5-2. Codes des catégories de log

Constante	Valeur	Texte
LOGCRITICAL	0x0001	CRITICAL
LOGERROR	0x0002	ERRROR
LOGWARNING	0x0004	WARNING
LOGCONFIG	0x0010	CONFIG
LOGVALUES	0x0020	VALUES
LOGIO	0x0100	IO
LOGDEVLIB	0x0200	DEVLIB
LOGDEBUG	0x1000	DEBUG
LOGDEBUGINOUT	0x2000	DEBUGINOUT
LOGDEBUGDUMP	0x4000	DEBUGDUMP
LOGALL	0x7337	N/A

Les constantes sont accessibles en utilisant le nom du module comme préfixe (elles ne sont pas exportées), par exemple `logger.LOGWARNING`. Les noms correspondant aux constantes sont

accessibles en utilisant le dictionnaire `logger.gcatnames` indexé sur les valeurs des constantes, et la liste des valeurs des constantes est accessible en utilisant la liste liste `logger.gcatconstants`. La constante `logger.LOGALL` est définie à partir des bits utilisés dans des constantes de catégorie.

Système de configuration

I. Références des Fonctions

catchmsg

Nom

`catchmsg` — chaine de message décrivant l'exception en cours.

Synopsis

Module: `logger`

La fonction `catchmsg` formate et retourne une chaîne décrivant l'exception en cours. Elle est utilisée dans les handlers catch afin de générer des logs contenant cette information.

Fonctionnement

```
def catchmsg () :
```

La fonction `catchmsg` récupère l'exception courante, en prend se représentation textuelle et la retourne (voir fonctions du module `traceback`).

catchtraceback

Nom

`catchtraceback` — chaine de message décrivant l'exception en cours.

Synopsis

Module: `logger`

La fonction `catchtraceback` formate et retourne une chaîne décrivant la pile d'appel pour l'exception en cours. Elle est utilisée dans les handlers catch afin de générer des logs contenant cette information.

Fonctionnement

```
def catchtraceback () :
```

La fonction `catchtraceback` récupère l'exception courante, en prend la représentation textuelle de la pile d'appel et la retourne (voir fonctions du module `traceback`).

dataconverter

Nom

`dataconverter` — création dynamique d'un convertisseur de données.

Synopsis

Module: `pyexp.DataConverterMod`

La fonction `dataconverter` se charge de créer une instance de la classe de conversion dont vous avez besoin. Elle se base sur l'argument qui lui est donné pour trouver quelle classe il faut instancier :

Tableau 1. Paramètres pour dataconverter

Paramètre	Classe instanciée
`"None"` ou `None`	pyexp.DataConvNoneMod.DataConvNone
`"linear"`	pyexp.DataconvLinearMod.DataConvLinear
`"classe"`	pyexp.classeMod.classe
`"module.classe"`	module.classe
`"package.module.classe"`	package.module.classe

Fonctionnement

```
def dataconverter (identifier) :
```

Une simple série de tests sur le paramètre ***`identifier`***, test sur la valeur, test sur la présence ou non d'un `"."` indiquant la présence d'un nom de module ou de package, et ensuite utilisation du module `dynbuildmod` qui se charge de créer une instance de la classe demandée (en chargeant le module de cette classe si ce n'est pas déjà fait).

datamanipulator

Nom

`datamanipulator` — création dynamique d'un manipulateur de données.

Synopsis

Module: `pyexp.DataManipulatorMod`

La fonction `datamanipulator` se charge de créer une instance de la classe de manipulation dont vous avez besoin. Elle se base sur l'argument qui lui est donné pour trouver quelle classe il faut instancier :

Tableau 1. Paramètres pour datamanipulator

Paramètre	Classe instanciée
`"int"` ou `"long"`, ou type entier ou long, ou valeur de type entier ou long	pyexp.DataManipIntMod.DataManipInt
`"float"`, ou type flottant, ou valeur de type flottant	pyexp.DataManipFloatMod.DataManipFloat
`"str"`, ou type chaîne, ou type chaîne Unicode	pyexp.DataManipStrMod.DataManipStr
`"[int]" ou "[long]"`	pyexp.DataManipInt1DMod.DataManipInt1D
`"[float]"`	pyexp.DataManipFloat1DMod.DataManipFloat1D
`"[str]"`	pyexp.DataManipStr1DMod.DataManipStr1D
`"[[int]]"` ou `"[[long]]"`	pyexp.DataManipInt2DMod.DataManipInt2D
`"[[float]]"`	pyexp.DataManipFloat2DMod.DataManipFloat2D
`"[[str]]"`	pyexp.DataManipStr2DMod.DataManipStr2D
`"classe"`	pyexp.classeMod.classe
`"module.classe"`	module.classe
`"package.module.classe"`	package.module.classe

Fonctionnement

```
def datamanipulator (identifier) :
```

Une simple série de tests sur le paramètre ***`identifier`***, test sur la valeur et sur son type, test sur la présence ou non d'un `"."` indiquant la présence d'un nom de module ou de package, et ensuite utilisation du module `dynbuildmod` qui se charge de créer une instance de la classe demandée (en chargeant le module de cette classe si ce n'est pas déjà fait).

disablelog

Nom

`disablelog` — désactivation d'une catégorie de logs.

Synopsis

Module: `logger`

La fonction `disablelog` permet de désactiver la génération de certains logs à partir de leur catégorie. Par défaut le filtrage s'applique à l'ensemble des logs, mais il peut se limiter aux logs d'une certaine source ou encore aux logs provenant de source non référencée.

Le premier paramètre *`category`* est un masque de bits construit à partir des constantes des catégories de log. Le second paramètre *`source`* est un nom de source de logs, par défaut il est à la valeur `"*"` pour indiquer que la fonction porte sur tous les logs d'où qu'ils viennent, il est possible d'utiliser la valeur `None` afin d'indiquer que la fonction ne doit porter que sur les logs provenant de source non référencée, et enfin il est possible de donner le nom de la source concernée par la fonction.

Il est à noter que pour des raisons de sécurité, la fonction empêche toute désactivation des logs critiques et des logs d'erreur, quelle que soit la source des logs.

Fonctionnement

```
def disablelog (category,source="*") :
```

La fonction `disablelog` se contente d'appeler la fonction `enablelog` en lui fournissant les indications de catégorie et de source qu'on lui a donné en paramètre, et en donnant 0 pour le paramètre *`enable`*.

dolog

Nom

`dolog` — fonction interne de génération d'un log.

Synopsis

Module: `logger`

La fonction `dolog` est appelée par les fonctions logxxx qui demandent la génération de logs de différentes catégories. Elle se charge de vérifier si le log est utile (s'il passe le filtrage), et si oui elle constitue un en-tête et elle formate le message avec les arguments avant d'envoyer le tout vers le système de log (`syslog`).

Fonctionnement

```
def dolog (source,category,fctname,format,args) :
```

La fonction `dolog` commence appliquer un masque de bits de filtrage par catégorie afin de vérifier que le log doit être généré ; en utilisant soit un masque spécifique à la source du log si celle-ci a été référencée, soit le masque général. Si le log ne doit pas être généré, la fonction retourne immédiatement. Ensuite, elle construit les différentes parties constituant l'en-tête côté logger (temps précis, thread, catégorie, source, fonction) et elle formate la chaîne de message si des arguments ont été donnés. Enfin elle génère un syslog en utilisant un niveau de log correspondant à la catégorie demandée, avec l'en-tête et le message formaté.

Tableau 1. Correspondance catégorie logger / niveau syslog

Catégorie logger	Niveau syslog
LOGCRITICAL	LOG_CRIT
LOGERROR	LOG_ERR
LOGWARNING	LOG_WARNING
LOGCONFIG	LOG_NOTICE
LOGVALUES	LOG_NOTICE
LOGIO	LOG_INFO
LOGDEVLIB	LOG_INFO
LOGDEBUG	LOG_DEBUG
LOGDEBUGINOUT	
LOGDEBUGDUMP	LOG_DEBUG
autres ou inconnu…	LOG_DEBUG

enablelog

Nom

`enablelog` — activation d'une catégorie de logs.

Synopsis

Module: `logger`

La fonction `enablelog` permet d'activer la génération de certains logs à partir de leur catégorie. Par défaut le filtrage s'applique à l'ensemble des logs, mais il peut se limiter aux logs d'une certaine source ou encore aux logs provenant de source non référencée.

Le premier paramètre *`category`* est un masque de bits construit à partir des constantes des catégories de log. Le second paramètre *`source`* est un nom de source de logs, par défaut il est à la valeur `"*"` pour indiquer que la fonction porte sur tous les logs d'où qu'ils viennent, il est possible d'utiliser la valeur `None` afin d'indiquer que la fonction ne doit porter que sur les logs provenant de source non référencée, et enfin il est possible de donner le nom de la source concernée par la fonction. Le troisième paramètre *`enable`* permet d'indiquer si l'on veut activer (passer la valeur 1) ou désactiver (passer la valeur 0) la génération des logs correspondant au masque (par défaut il est à 1, ce qui correspond à la sémantique de la fonction lorsque l'on ne donne qu'un ou deux paramètres).

Il est à noter que pour des raisons de sécurité, la fonction empêche toute désactivation des logs critiques et des logs d'erreur, quelle que soit la source des logs.

Fonctionnement

```
def enablelog (category,source="*",enable=1) :
```

La fonction `enablelog` commence par mettre à jour le masque de catégorie afin d'empêcher la désactivation des logs critiues ou d'erreur. Ensuite, suivant la source indiquée, elle met à jour les différents masques mémorisés par le module (marque général par défaut, masques liés aux sources identifiées).

logcategories

Nom

`logcategories` — retourne un filtre de catégories, celui par défaut ou un pour une source de log.

Synopsis

Module: `logger`

La fonction `logcategories` retourne le masque de bits de filtrage de catégorie, soir pour le filtre par défaut (si l'on indique une source à None ou une source non référencée), soit pour une source référencée.

Fonctionnement

```
def logcategories (source=None) :
```

La fonction `logcategories` vas simplement chercher la valeur du masque de bits de filtrage, soit dans le dictionnaire des sources référencées, soit dans la globale du filtre par défaut courant.

logconfig

Nom

`logconfig` — génération d'un log de catégorie configuration.

Synopsis

Module: `logger`

La fonction `logconfig` demande la génération d'un log de configuration, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour tout ce qui a trait aux données de configuration d'où elles viennent, leurs valeurs, si elles sont modifiées… Pour les paramètres ***`source`*** et ***`fctname`***, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logconfig (source,fctname,format,*args) :
```

La fonction `logconfig` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGCONFIG`. Pour les paramètres ***`source`*** et ***`fctname`***, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

logcritical

Nom

`logcritical` — génération d'un log d'erreur critique.

Synopsis

Module: `logger`

La fonction `logcritical` demande la génération d'un log d'erreur critique, qui se traduit normalement par un log effectif (les logs de ce niveau ne peuvent pas être désactivés). On l'utilise lorsque se produit une erreur importante pouvant entraîner la perte de données ou l'arrêt de l'expérience (par exemple un disque de données plein, un périphérique essentiel qui ne répond plus…). Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logcritical (source,fctname,format,*args) :
```

La fonction `logcritical` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGCRITICAL`. Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

logdebug

Nom

`logdebug` — génération d'un log de débogage.

Synopsis

Module: `logger`

La fonction `logdebug` demande la génération d'un log de débogagae, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour les informations de débogage importantes (voir aussi `logdebuginout` et `logdebugdump`). Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logdebug (source,fctname,format,*args) :
```

La fonction `logdebug` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGDEBUG`.

logdebuginout

Nom

`logdebuginout` — génération d'un log d'indication d'entrée/sortie de méthode/fonction.

Synopsis

Module: `logger`

La fonction `logdebuginout` demande la génération d'un log de trace des entrées/sorties de code, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour les traces des entrées et sorties des fonctions et méthodes, des interceptions d'exceptions, de leur retransmission… Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logdebuginout (source,fctname,format,*args) :
```

La fonction `logdebuginout` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGDEBUGINOUT`.

logdebugdump

Nom

`logdebugdump` — génération d'un log de débogage bas niveau.

Synopsis

Module: `logger`

La fonction `logdebugdump` demande la génération d'un log de débogage bas niveau, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour les informations de débogage de bas niveau, affichage de données, signalisation de passage… Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logdebugdump (source,fctname,format,*args) :
```

La fonction `logdebugdump` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGDEBUGDUMP`.

logdevlib

Nom

`logdevlib` — génération d'un log d'indication de configuration.

Synopsis

Module: `logger`

La fonction `logdevlib` demande la génération d'un log de configuration, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour les traces des appels à des librairies de gestion de périphériques (hors librairies de communication style série, gpib, réseau qui sont en logio). Pour les paramètres ***`source`*** et ***`fctname`***, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logdevlib (source,fctname,format,*args) :
```

La fonction `logdevlib` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGDEVLIB`.

logerror

Nom

`logerror` — génération d'un log d'erreur.

Synopsis

Module: `logger`

La fonction `logerror` demande la génération d'un log d'erreur, qui se traduit normalement par un log effectif (les logs de ce niveau ne peuvent pas être désactivés). On l'utilise lorsque se produit une erreur notable, éventuellement suffisante pour stopper un traitement en cours et remonter par une exception. Pour les paramètres ***`source`*** et ***`fctname`***, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logerror (source,fctname,format,*args) :
```

La fonction `logerror` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGERROR`.

logio

Nom

`logio` — génération d'un log d'indication d'entrées/sorties.

Synopsis

Module: `logger`

La fonction `logio` demande la génération d'un log de suivi de valeurs, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour les trace de communications avec un périphérique style série, gpib, réseau. Cela inclus les traces des appels aux fonctions d'établissement de la communication, d'attente, ainsi que tout ce qui est données écrites ou lues sur le périphérique. Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logio (source,fctname,format,*args) :
```

La fonction `logio` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGVALUES`.

logvalues

Nom

`logvalues` — génération d'un log de modification de valeurs.

Synopsis

Module: `logger`

La fonction `logvalues` demande la génération d'un log de suivi de valeurs, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour tout ce qui est modification de valeurs importantes. Typiquement, les modifications de valeurs dans les grandeurs physique de pyexp. Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logvalues (source,fctname,format,*args) :
```

La fonction `logvalues` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGVALUES`.

logwarning

Nom

`logwarning` — génération d'un log d'alerte.

Synopsis

Module: `logger`

La fonction `logwarning` demande la génération d'un log d'alerte, qui peut être ou ne pas être effectif suivant le filtrage en cours. On l'utilise pour indiquer un problème suite auquel l'exécution a pû continuer (problème annecdotique, ou action de correction). Par exemple, lorsque l'on intercepte une exception et qu'on la corrige, on génère un log de warning. Pour les paramètres *`source`* et *`fctname`*, il est possible de passer la valeur `None` lorsqu'ils ne sont pas connus. Pour la chaîne de formatage, il est possible de lui passer des valeurs à formater comme paramètres supplémentaires.

Fonctionnement

```
def logwarning (source,fctname,format,*args) :
```

La fonction `logwarning` se contente d'appeler la fonction `dolog` interne au module `logger`, en lui donnant la catégorie de log `LOGWARNING`.

registeredlogsources

Nom

`registeredlogsources` — liste des sources de log enregistrées.

Synopsis

Module: `logger`

La fonction `registeredlogsources` retourne la liste des noms de sources de log enregistrés.

Fonctionnement

```
def registeredlogsources () :
```

La fonction `registeredlogsources` récupère simplement la liste des noms enregistrés dans le dictionnaire des sources référencées. Il est possible de récupérer le filtre des catégories d'une source en utilisant la fonction `logcategories`.

registerlogsource

Nom

`registerlogsource` — enregistrement d'une source de log.

Synopsis

Module: `logger`

La fonction `registerlogsource` permet de référencer un nom de source de log. Elle sera typiquement utilisée lors des phases d'initialisation.

Fonctionnement

```
def registerlogsource (source) :
```

La fonction `def registerlogsource (source) :` se contente de tester si la source est déjà référencée dans un dictionnaire des sources, et si ce n'est pas le cas de créer une entrée en y associant la valeur du filtre des catégorie courant par défaut. Il est possible de récupérer la liste des sources enregistrées en utilisant la fonction `registeredlogsources`.

thetime

Nom

`thetime` — retourne le temps absolu de façon précise.

Synopsis

Module: `logger`

La fonction `thetime` retourne une valeur similaire à la fonction `time` (module `time`), mais en donnant un temps au niveau de précision de la fonction `clock` (module `time` aussi). Typiquement, la fonction `time` donne une résolution au centième de seconde, alors que `thetime` descend à quelques dizaines de micro-secondes. Attention, cette fonction ne fait pas partie des symboles exportés, et doit donc être accédée en la préfixant du nom du module.

Fonctionnement

```
def thetime () :
```

Le module, à son chargement, stocke dans deux globales les valeurs de `time` et de `clock`. Ensuite, la fonction `thetime` se base sur l'évolution du temps mesurée par la fonction `clock`, en y ajoutant la datation absolue mémorisée pour time.

thetimestr

Nom

`thetimestr` — formate le temps absolu de façon précise.

Synopsis

Module: `logger`

La fonction `thetimestr` prend une datation (a priori retournée par la fonction `thetime` du même module), et en fait un formatage sous forme de chaîne, qui soit non seulement précis, mais en plus utilisable pour des en-têtes triables. Si aucune datation ne lui est donnée, la fonction vas d'elle-même chercher la valeur courante de `thetime`. Attention, cette fonction ne fait pas partie des symboles exportés, et doit donc être accédée en la préfixant du nom du module.

Fonctionnement

`def thetimestr (t=None) :`

La fonction `thetimestr` récupère la datation courante si aucune datation ne lui a été donné, puis formate une chaîne correspondant à cette datation avec une précision à la dix-millième de seconde.

```
>>> print logger.thetimestr()

2001-12-11 10:57:06.5451
```

II. Références des Classes

Actor

Nom

`Actor` — Classe abstraite pour la hiérarchie des acteurs de PyExp (grandeurs, devices, groupes).

Synopsis

Module : `pyexp.ActorMod`

Cette classe fournit le systèmes de routage pour les préparations et les exécutions des requêtes sur les acteurs, la gestion du graphe des acteurs, le processus d'initialisation des acteurs, bref une grande partie du noyau de base de PyExp. Il n'y a pas d'instance directe de cette classe (aucune utilité).

Contrôle d'accès

Un petit système de contrôle d'accès, permettant de bloquer la création d'un acteur ou encore sa manipulation directe par un utilisateur, a commencé a être mis en place dans PyExp (cf informations profilebuild et profileview de la configuration), mais n'est pas encore complètement finalisé (source de l'identification des utilisateurs et de leurs profils).

Configuration

A chaque objet acteur correspond une entité de configuration contenant les informations nécessaires pour le créer et l'initialiser. Chaque classe d'acteurs - aux différents niveaux d'héritage - définit une ou plusieurs sections de configuration, dans lesquelles elle stocke ses informations propres.

Tous les acteurs de PyExp (ie. les instances des sous-classes de Actor) possèdent une méthode de configuration do_loadconf, appelée directement par les fonctions de PyExp lorsqu'un objet doit être initialisé ou réinitialisé. Cette méthode est en général redéfinie dans les sous-classes, qui appellent la méthode héritée de la classe parente afin qu'elle puisse charger ses paramètres de configuration.

Section ACTOR

Pour la classe `Actor`, les informations de configuration sont stockées dans la section ACTOR. On pourra trouver les noms de valeurs suivants :

actclass

Path de la classe d'acteur à instancier. Par exemple `pyexp.PhysElementMod.PhysElement`.

profilebuild

Liste de noms de profils utilisateurs autorisés à instancier cet acteur. On utilise `["*"]` pour donner l'autorisation à tout le monde.

profileview

Liste de noms de profils utilisateurs autorisés à accéder à cet acteur de façon directe (appel à getactor par l'utilisateur). Par exemple `["admin","respexp"]` pour ne permettre qu'à l'administrateur et au responsable d'expérience d'de manipuler directement cet acteur. On utilise `["*"]` pour donner l'autorisation à tout le monde.

upslots

Liste des noms des upslots de l'acteur. Par exemple `["axis1", "axis2", "axis3", "adc1, "adc2", "io1", "io2", "io3"]`. Les classes d'acteurs définissent généralement les noms standards des upslots qu'elles gèrent - voir leurs documentations respectives.

downslots

Liste des noms des downslots de l'acteur. Par exemple `["leftaxis", "rightaxis"]`. Les classes d'acteurs définissent généralement les noms standards des downslots qu'elles gèrent - voir leurs documentations respectives.

description

Simple chaîne de caractères décrivant l'acteur (son rôle) en une ligne.

wantsplitmeasure

Indicateur booléen (0|1) permettant à PyExp de savoir s'il faut pour cet acteur séparer les actions `MEASURE` en trois actions correspondant aux étapes de a mesure (`STARTMEASURE` + `TIMECOUNT` + `STOPMEASURE`). Cet indicateur peut être modifié dynamiquement en appelant la méthode `wantsplitmeasure` de la classe `Actor`.

parallelexecute

Indicateur booléen (0|1) permettant à PyExp de savoir s'il faut pour cet acteur exécuter en parallèle les requêtes destinées à différentes méthodes (elles sont par défaut exécutées de façon séquentielle, méthode par méthode).

Section DOWNLINKS

Afin de décrire les liens entre les acteurs, il existe en général une section DOWNLINKS, dans laquelle chaque nom de valeur est un nom de downslot (listé dans [ACTOR]downslots), et chaque valeur décrit les liens partant de ce downslot. Les valeurs peuvent être :

- Un tuple contenant deux chaînes indiquant le nom de l'acteur lié, et le nom du upslot par lequel arrive le lien sur cet acteur.
- Un chaîne indiquant le nom de l'acteur lié ; le nom d'upslot `"value"` est alors automatiquement utilisé (upslot standard pour les grandeurs).
- Une liste composée de tuples ou chaînes comme décrites aux deux points précédents, il s'agit alors de liens multiples.

Exemples :

```
leftaxis=("axisdev4","axis3")
rightaxis="bragg"
movers=[("axisdev2","axis5"),("engine","speed")]
```

Les liens ont en général toujours un acteur grandeur à une des extrémités (on n'a normalement pas de lien direct device ⇔ device ou device ⇔ groupe ou groupe ⇔ groupe).

Variables Membres

Voici les différentes variables membres des objets de la classe Actor. Elles sont toutes privées, certaines étant accessibles via des méthodes.

__name

Chaîne de caractères, nom identifiant l'acteur. Cette variable est accessible en lecture par la méthode `name`.

__confurl

Chaîne de caractères, URL de la source de configuration pour l'acteur.

__confname

Chaîne de caractères, nom d'entité dans la source de configuration pour l'acteur.

__mutex

Comme son nom l'indique, objet système d'exclusion mutuelle pour la gestion des accès concurrents à l'acteur. Cette variable est utilisée lors de l'utilisation des méthodes `acquire` et `release`.

__membersready

Variable booléenne mémorisant la valeur de retour de l'appel à do_resetmembers. Normalement 1 si tout s'est bien passé, et 0 en cas d'erreur.

__configready

Variable booléenne mémorisant la valeur de retour de l'appel à do_loadconf. Normalement 1 si tout s'est bien passé, et 0 en cas d'erreur.

__linksready

Variable booléenne mémorisant la valeur de retour de l'appel à do_setuplinks. Normalement 1 si tout s'est bien passé, et 0 en cas d'erreur.

__devlinkready

Variable booléenne mémorisant la valeur de retour de l'appel à do_connectdevice. Normalement 1 si tout s'est bien passé, et 0 en cas d'erreur.

__devconfready

Variable booléenne mémorisant la valeur de retour de l'appel à do_configdevice. Normalement 1 si tout s'est bien passé, et 0 en cas d'erreur.

Il y a de plus une série de variables internes de débug, éfinie uniquement si la globale `CHECK_INHERIT` est vraie. Ces variables sont initialisées à 0, et positionnées à 1 lors de l'appel de certaines méthodes de la classe Actor. Elles permettent de vérifier que les sous-classes font bien appel aux méthodes héritées des classes parentes.

__ckresetmembers

Contrôle l'appel à do_resetmembers.

__ckloadconf

Contrôle l'appel à do_loadconf.

__cksetuplinks

Contrôle l'appel à do_setuplinks.

__ckconfiglinks

INUTILISE! (Contrôle l'appel à do_configlinks - mais l'appel de celui-ci est commenté).

DataConverter

Nom

`DataConverter` — Classe abstraite pour la hiérarchie des classes de convertisseurs de données.

Synopsis

Module: `pyexp.DataConverterMod`

Cette classe donne le cadre dans lequel les sous classes de conversion doivent être définies afin d'être utilisables de façon transparente par pyexp. Elle n'implémente **aucune** fonction de conversion particulière.

Elle assure une certaine sécurité concernant les valeurs données en entrée dans les méthodes de conversion. En effet, certaines opérations de conversion peuvent être destructrices pour la valeur source, d'autres transmettre des valeurs partagées entre source et destination. Dans ce cas le convertisseur doit s'être caractérisé comme tel, et les fonctions de haut niveau de conversion se chargeront -si c'est nécessaire- de dupliquer automatiquement les valeurs avant de les envoyer vers le code effectif de conversion et de retourner la valeur finale.

Configuration

Les objets de la classe DataConverter et de ses sous-classes chargent leur configuration à partir de leurs méthodes `do_loadconfig` (qui ne doivent pas être appelée directement, il faut passer par la méthode `loadconfig` qui se charge d'appeler `do_loadconfig` dans le bon contexte).

Section CONVERTER

Par défaut la section décrivant les paramètres du convertisseur est nommée CONVERTER, mais cette classe peut être utilisée dans d'autres contextes, et un autre nom de section peut être donné à la méthode loadconfig. Dans cette section de configuration on pourra trouver les noms de valeurs suivants :

usermodifiable

> Valeur booléenne (0|1) permettant d'indiquer au convertisseur si un utilisateur a le droit de modifier des paramètres de conversion en appelant directement les méthodes du convertisseur. Valeur stockée dans la variable membre `__usermodifiable`. En cas d'absence dans la section de configuration, "l'ancienne" valeur de `__usermodifiable` est utilisée. Cette valeur est la seule valeur lue dans la méthode `loadconfig`, tous les autres étant lues dans les méthodes `do_loadconfig`.

alwaysduplicate

> Valeur booléenne (0|1) permettant d'indiquer au convertisseur qu'il faut systématiquement effectuer une copie des valeurs avant d'effectuer les conversions. Valeur stockée dans la variable membre `__alwaysduplicate`. En cas d'absence lors du chargement à partir de la configuration, la valeur 1 est utilisée pour plus de sécurité.

hnotbemodified

Valeur booléenne (0|1) permettant d'indiquer au convertisseur que les valeurs exprimées en unité 'high' et passées en paramètre ne doivent pas être modifiées lors de la conversion. Valeur stockée dans la variable membre `__hnotbemodified`. En cas d'absence lors du chargement à partir de la configuration, la valeur 1 est utilisée pour plus de sécurité.

lnotbemodified

Valeur booléenne (0|1) permettant d'indiquer au convertisseur que les valeurs exprimées en unité 'low' et passées en paramètre ne doivent pas être modifiées lors de la conversion. Valeur stockée dans la variable membre `__lnotbemodified`. En cas d'absence lors du chargement à partir de la configuration, la valeur 1 est utilisée pour plus de sécurité.

Variables Membres

Toutes les variables membres de cette classe sont privées, et sont utilisées pour les comportements par défaut.

__usermodifiable

Booléen, indicateur que les paramètres de conversion sont modifiables par l'utilisateur. Initialisé à 0 lors de la construction d'un `DataConverter`. Chargé lors de la configuration à partir de la valeur booléenne stockée sous le nom `usermodifiable` dans la configuration (voir section Configuration). Consultable et modifiable avec la méthode `usermodifiable`. Valeur testée dans la méthode `checkusermodifiable` afin de savoir s'il faut lever une exception ou non.

__alwaysduplicate

Booléen, indicateur que le convertisseur doit toujours effectuer une copie des valeurs passées en entrée dans les fonctions de conversion. Utile par exemple lorsque l'on a à faire à un convertisseur qui se contente de transmettre la valeur donnée sans y toucher, en évitant qu'une valeur disponible en bas niveau soit ensuite la même utilisée par tous les appelants en haut niveau (tout le monde aurait alors une référence sur le même objet, sans savoir qu'il est partagé, et certains pourraient le modifier à l'insu des autres). Initialisé à 1 lors de la construction d'un `DataConverter`. Chargé lors de la configuration à partir de la valeur booléenne stockée sous le nom `alwaysduplicate` dans la configuration (voir section Configuration). Consultable et modifiable avec la méthode `alwaysduplicate`. Valeur testée dans les méthodes `h2l` et `l2h`afin de vérifier si la duplication est forcée.

__hash2l

Booléen, indicateur que le convertisseur supporte la conversion d'unité 'high' vers l'unité 'low'. Initialisé à 0 lors de la construction d'un `DataConverter`. Normalement modifié par les sous-classes en utilisant le paramètre *`hash2l`* de la méthode `characterize`. Valeur testée dans la méthode `h2l` afin de vérifier si la conversion est supportée.

__h2lmodifysource

Booléen, indicateur que la conversion de l'unité 'high' vers l'unité 'low' provoque une modification de la valeur en unité 'high' donnée en paramètre. Initialisé à 1 lors de la construction d'un `DataConverter`. Normalement modifié par les sous-classes en utilisant le paramètre *`h2lmodsrc`* de la méthode `characterize`. Utilisé dans la méthode `h2l` pour savoir s'il faut effectuer une recopie de la valeur en paramètre avant de lancer la conversion.

__hnotbemodified

Booléen, indicateur que la valeur donnée en unité high ne doit pas être modifiée (i.e. en cas de risque de modification, elle doit être dupliquée et la copie doit être utilisée). Initialisé à 1 lors de la construction d'un `DataConverter`. Utilisé dans la méthode `h2l` pour savoir s'il faut effectuer une recopie de la valeur en paramètre avant de lancer la conversion. Cet indicateur peut être consulté et modifié par l'utilisateur avec la méthode `hnotbemodified`.

__hmanip

Manipulateur de données (instance d'une sous-classe de `DataManipulator`), utilisé lorsqu'il y a besoin de manipuler les valeurs exprimées en unité high (principalement pour les copies, mais utilisable pour les lectures et écritures). Initialisé avec un `DataManipCopy` lors de la construction d'un `DataConverter`. Normalement modifié par les sous-classes en utilisant le paramètre ***`hmanip`*** de la méthode `characterize`. Utilisé dans la méthode `h2l` lorsqu'il faut effectuer une recopie de la valeur en paramètre avant de lancer la conversion. Accessible par l'utilisateur avec la méthode `hmanip`.

__hasl2h

Booléen, indicateur que le convertisseur supporte la conversion d'unité 'low' vers l'unité 'high'. Initialisé à 0 lors de la construction d'un `DataConverter`. Normalement modifié par les sous-classes en utilisant le paramètre ***`hasl2h`*** de la méthode `characterize`. Valeur testée dans la méthode `l2h` afin de vérifier si la conversion est supportée.

__l2hmodifysource

Booléen, indicateur que la conversion de l'unité 'low' vers l'unité 'high' provoque une modification de la valeur en unité 'low' donnée en paramètre. Initialisé à 1 lors de la construction d'un `DataConverter`. Normalement modifié par les sous-classes en utilisant le paramètre ***`l2hmodsrc`*** de la méthode `characterize`. Utilisé dans la méthode `l2h` pour savoir s'il faut effectuer une recopie de la valeur en paramètre avant de lancer la conversion.

__lnotbemodified

Booléen, indicateur que la valeur donnée en unité low ne doit pas être modifiée (i.e. en cas de risque de modification, elle doit être dupliquée et la copie doit être utilisée). Initialisé à 1 lors de la construction d'un `DataConverter`. Utilisé dans la méthode `l2h` pour savoir s'il faut effectuer une recopie de la valeur en paramètre avant de lancer la conversion. Cet indicateur peut être consulté et modifié par l'utilisateur avec la méthode `lnotbemodified`.

__lmanip

Manipulateur de données (instance d'une sous-classe de `DataManipulator`), utilisé lorsqu'il y a besoin de manipuler les valeurs exprimées en unité low (principalement pour les copies, mais utilisable pour les lectures et écritures). Initialisé avec un `DataManipCopy` lors de la construction d'un `DataConverter`. Normalement modifié par les sous-classes en utilisant le paramètre ***`lmanip`*** de la méthode `characterize`. Utilisé dans la méthode `l2h` lorsqu'il faut effectuer une recopie de la valeur en paramètre avant de lancer la conversion. Accessible par l'utilisateur avec la méthode `lmanip`.

__init__

```
x = DataConvXXX ()
```

```
def __init__ (self) :
```

La méthode `__init__` est appelée automatiquement lors de la construction d'un `DataConverter`. Elle doit être appelée explicitement par les constructeurs des sous-classes de `DataConverter` dans leur méthode `__init__` propre, et généralement au début du code de ces méthodes, qui doivent ensuite appeler la méthode `characterize` afin de spécifier leurs caractéristiques propres pour permettre un fonctionnement optimal.

La méthode __init__ de DataConverter se charge simplement d'initialiser les variables membres de cette classe (__usermodifiable, __alwaysduplicate, __hash2l, __h2lmodifysource, __hnotbemodified, __hmanip, __hasl2h, __l2hmodifysource, __lnotbemodified, __lmanip) avec des valeurs par défaut.

h2l

```
conv.h2l (valeur "high" [,forçage duplication]) → valeur "low"
```

```
def h2l (self, value, forcedupl=0) :
```

Méthode de conversion d'une valeur exprimée en unité 'high' vers une valeur exprimée en unité 'low'. Outre la valeur à convertir, la méthode prend un paramètre optionnel ***`forcedupl`***, qui permet de forcer la duplication de la valeur en entrée lors d'une demande de conversion (même si les paramètres de fonctionnement automatique n'auraient pas conduit à cette duplication) lorsqu'elle est mise à 1 (exactement lorsqu'elle est >0), ou de forcer l'utilisation de la valeur originale dans la conversion lorsqu'elle est mise à -1 (exactement lorsqu'elle est <0).

Il existe des synonymes de cette méthode, définis pour des raisons de sémantique dans certains contextes d'utilisation : `u2d` (pour les relations unité utilisateur → unité data des grandeurs), `e2m` (pour les relations unité codeur → unité moteur des grandeurs axes), `p2r` (pour les relations valeurs pseudo → valeurs réelles des groupes de grandeurs).

Cette méthode n'effectue pas elle-même la conversion, elle appelle la méthode `do_h2lconversion` qui est chargée d'effectuer les calculs.

do_h2lconversion

```
self.do_h2lconversion (valeur "high") → valeur "low"
```

```
def do_h2lconversion (self, value) :
```

Les sous-classes qui implémentent une conversion de l'unité 'high' vers l'unité 'low' doivent redéfinir cette méthode afin d'y mettre leur code de conversion. L'implémentation par défaut lève une exception avec une indication d'erreur.

l2h

```
conv.l2h (valeur "low" [,forçage duplication]) → valeur "high"
```

```
def l2h (self, value, forcedupl=0) :
```

Méthode de conversion d'une valeur exprimée en unité 'low' vers une valeur exprimée en unité 'high'. Outre la valeur à convertir, la méthode prend un paramètre optionnel ***`forcedupl`***, qui permet de forcer la duplication de la valeur en entrée lors d'une demande de conversion (même si les paramètres de fonctionnement automatique n'auraient pas conduit à cette duplication) lorsqu'elle est mise à 1 (exactement lorsqu'elle est >0), ou de forcer l'utilisation de la valeur originale dans la conversion lorsqu'elle est mise à -1 (exactement lorsqu'elle est <0).

Il existe des synonymes de cette méthode, définis pour des raisons de sémantique dans certains contextes d'utilisation : `d2u` (pour les relations unité data → unité utilisateur des grandeurs), `m2e` (pour les relations unité moteur → unité codeur des grandeurs axes), `r2p` (pour les relations valeurs réelles → valeurs pseudo des groupes de grandeurs).

Cette méthode n'effectue pas elle-même la conversion, elle appelle la méthode `do_l2hconversion` qui est chargée d'effectuer les calculs.

do_l2hconversion

`self.do_l2hconversion (valeur "low") → valeur "high"`

`def do_l2hconversion (self, value) :`

Les sous-classes qui implémentent une conversion de l'unité 'low' vers l'unité 'high' doivent redéfinir cette méthode afin d'y mettre leur code de conversion. L'implémentation par défaut lève une exception avec une indication d'erreur.

convdirection

`conv.convdirection (valeur "low" ou valeur "high") → constante CONVDIR`

`def convdirection (self, highvalue=None, lowvalue=None) :`

Méthode permettant de connaître, en un point donné, le sens croissant/décroissant de la fonction de conversion. Cela permet de savoir si, lorsqu'une des deux valeurs croît, l'autre vas croître ou décroître. La méthode prend en paramètre une valeur "high" ou "low", mais **pas les deux en même temps**, qui doit être indiquée comme paramètre nommé (c'est tout du moins fortement conseillé). Pour la valeur donnée en paramètre, la méthode retourne une constante parmis :

`CONVDIR.SAME` pour indiquer que les deux valeurs évoluent dans le même sens
`CONVDIR.INVERTED` pour indiquer que les deux valeurs évoluent en sens inverse
`CONVDIR.UNKNOWN` pour indiquer que la méthode ne sait pas dans quel sens les deux valeurs évoluent

Cette méthode n'effectue pas elle-même la conversion, elle appelle la méthode `do_convdirection` qui est chargée d'estimer la direction de la conversion aux alentours du point.

do_convdirection

`self.do_convdirection (valeur "low" ou valeur "high") → constante CONVDIR`

```
def do_convdirection (self, highvalue,lowvalue) :
```

Les sous-classes qui peuvent connaître le sens des conversions aux alentours des valeurs, doivent implémenter cette méthode. Le paramètre highvalue/lowvalue qui n'a pas été donné lors de l'appel est à `None`. Le comportement par défaut de cette méthode est de retourner `CONVDIR.UNKNOWN`.

characterize

`self.characterize ([hash2l=booléen] [,hasl2h=booléen] [,h2lmodsrc=booléen] [,l2hmodsrc=booléen] [,hmanip=manipulateur de données] [,lmanip=manipulateur de données])`

```
def characterize (self, hash2l=None, hasl2h=None, h2lmodsrc=None, l2hmodsrc=None, hmanip=None,l manip=None) :
```

Fixe les caractéristiques du convertisseur afin que les automatismes de conversion soient optimisés et sécurisés, et puissent fonctionner. Chaque paramètre de cette méthode donne une caractéristique à associée au convertisseur. Cette méthode est généralement appelée lors de la construction des convertisseurs. Elle peut toutefois aussi être appelée à d'autres moments afin de changer dynamiquement un attribut de caractérisation d'un convertisseur (les paramètres non précisés ou précisés à None ne génèrent aucune modification dans les variables membres de l'objet).

hash2l

> De type booléen, indicateur de support par le convertisseur de la conversion de l'unité 'high' vers l'unité 'low'. Si ce paramètre n'est pas à `None`, il est stocké dans la variable membre `__hash2l`.

hasl2h

> De type booléen, indicateur de support par le convertisseur de la conversion de l'unité 'low' vers l'unité 'high'. Si ce paramètre n'est pas à `None`, il est stocké dans la variable membre `__hasl2h`.

h2lmodsrc

> De type booléen, indicateur que la conversion de l'unité 'high' vers l'unité 'low' modifie la valeur source. Si ce paramètre n'est pas à `None`, il est stocké dans la variable membre `__h2lmodifysource`.

l2hmodsrc

De type booléen, indicateur que la conversion de l'unité 'low' vers l'unité 'high' modifie la valeur source. Si ce paramètre n'est pas à `None`, il est stocké dans la variable membre `__l2hmodifysource`.

hmanip

De type sous-classe de `DataManipulator`, manipulateur pour les valeurs exprimées en unité 'high'. Utilisé par les automatismes de conversion lorsqu'il faut dupliquer les valeurs en unité 'high' afin d'éviter qu'elles soient modifiées. Si ce paramètre n'est pas à `None`, il est stocké dans la variable membre `__hmanip`.

lmanip

De type sous-classe de `DataManipulator`, manipulateur pour les valeurs exprimées en unité 'low'. Utilisé par les automatismes de conversion lorsqu'il faut dupliquer les valeurs en unité 'low' afin d'éviter qu'elles soient modifiées. Si ce paramètre n'est pas à `None`, il est stocké dans la variable membre `__lmanip`.

usermodifiable

`conv.usermodifiable ([booléen]) → booléen`

`def usermodifiable (self, usermodif=None) :`

Accès à - et modification de - l'indicateur d'autorisation de modification des paramètres de conversion. Si ***`usermodif`*** est à `None`, l'indicateur stocké dans la variable membre `__usermodifiable` n'est pas modifié et sa valeur est simplement retournée. Si ***`usermodif`*** n'est pas à `None`, sa représentation booléenne est stockée dans la variable membre `__usermodifiable`, dont l'ancienne valeur est retournée.

alwaysduplicate

`conv.alwaysduplicate ([booléen]) → booléen`

`def alwaysduplicate (self, alwdupl=None) :`

Accès à - et modification de - l'indicateur de forçage de duplication des valeurs avant conversion. Si ***`alwdupl`*** est à `None`, l'indicateur stocké dans la variable membre `__alwaysduplicate` n'est pas modifié et sa valeur est simplement retournée. Si ***`alwdupl`*** n'est pas à `None`, sa représentation booléenne est stockée dans la variable membre `__alwaysduplicate`, dont l'ancienne valeur est retournée.

hnotbemodified

`conv.hnotbemodified ([booléen]) → booléen`

`def hnotbemodified (self, ensure=None) :`

Accès à - et modification de - l'indicateur d'autorisation de modification des valeurs 'high' passées lors des conversions. Si ***`ensure`*** est à `None`, l'indicateur stocké dans la variable membre `__hnotbemodified` n'est pas modifié et sa valeur est simplement retournée. Si ***`ensure`*** n'est pas à `None`, sa valeur est stockée dans la variable membre `__hnotbemodified`, dont l'ancienne valeur est retournée.

lnotbemodified

`conv.lnotbemodified ([booléen]) → booléen`

`def lnotbemodified (self, ensure=None) :`

Accès à - et modification de - l'indicateur d'autorisation de modification des valeurs 'low' passées lors des conversions. Si *`ensure`* est à `None`, l'indicateur stocké dans la variable membre `__lnotbemodified` n'est pas modifié et sa valeur est simplement retournée. Si *`ensure`* n'est pas à `None`, sa valeur est stockée dans la variable membre `__lnotbemodified`, dont l'ancienne valeur est retournée.

checkusermodifiable

`self.checkusermodifiable ([nom de paramètre])`

`def checkusermodifiable (self,paramname="any parameter") :`

Vérifie si les paramètres de conversion du DataConverter sont modifiables par l'utilisateur (teste la variable membre `__usermodifiable`). Si c'est le cas, la méthode retourne simplement. Si ce n'est pas le cas, la méthode lève une exception de la classe `RuntimeError`, avec une information comme quoi le paramètre de conversion *`paramname`* n'est pas modifiable. Cette méthode est typiquement appelée par les sous-classes dans leurs méthodes propres, afin de vérifier que l'utilisateur peur modifier un paramètre de conversion avant de le modifier effectivement.

loadconfig

`conv.loadconfig (entité de configuration [,nom de section])`

`def loadconfig (self,conf,sect="CONVERTER") :`

Demande au convertisseur de charger ses paramètres de fonctionnement. Pour qu'elle puisse trouver les paramètres de fonctionnement, il faut fournir à la méthode `loadconfig` une référence sur un objet Config (cf. module confmod) et le nom d'une section (on utilisera généralement la section de nom "`CONVERTER`", qui est la valeur par défaut de ce paramètre).

Le `loadconfig` désactive temporairement le blocage des modifications par l'utilisateur (on considère que cette fonction sera principalement utilisée lors des initialisations, de plus l'appelant doit tout de même disposer d'un objet Config avec la section adéquate, ce qui limite les risques d'appel "par erreur"). Ensuite `loadconfig` appelle la méthode `do_loadconfig`, qui doit être redéfinie dans les sous-classes qui supportent le chargement de leurs paramètres à partir du système de configuration. Enfin loadconfig recharge l'indicateur de blocage des modifications par l'utilisateur en utilisant la valeur nommée "`usermodifiable`" dans la section et l'entité de configuration donnés en paramètres.

do_loadconfig

`self.do_loadconfig (entité de configuration, nom de section)`

`def do_loadconfig (self,conf,sect) :`

C'est la méthode que les sous-classes de `DataConverter` doivent redéfinir pour supporter le chargement de leurs paramètres à partir du système de configuration. L'implémentation d'une méthode do_loadconfig doit commencer par appeler l'implémentation héritée de la classe parente (afin que celle-ci puisse charger les paramètres à son niveau). Elle doit ensuite charger ses paramètres propres (elle peut utiliser ses fonctions de modification des paramètres afin de bénéficier des contrôles de valeurs qu'elles fournissent). En cas d'erreur, l'exception doit être remontée, et il est bienvenue d'avoir stocké les anciens paramètres et de les remettre en place (ou sinon de générer des exceptions aux prochaines demandes de conversion tant que des paramètres corrects ne sont pas mis en place).

storeconfig

`conv.storeconfig (entité de configuration [,nom de section])`

`def storeconfig (self,conf,sect="CONVERTER") :`

Demande au convertisseur de stocker ses paramètres de fonctionnement. Pour qu'elle puisse sauvegarder les paramètres de fonctionnement, il faut fournir à la méthode `loadconfig` une référence sur un objet Config (cf. module confmod) et le nom d'une section (on utilisera généralement la section de nom "`CONVERTER`", qui est la valeur par défaut de ce paramètre).

Les sous-classes de `DataConverter` qui ont des paramètres de fonctionnement à sauvegarder doivent redéfinir la méthode `do_storeconfig`.

do_storeconfig

`self.storeconfig (entité de configuration, nom de section)`

`def do_storeconfig (self,conf,sect) :`

Méthode effectuant le stockage des paramètres de fonctionnement du convertisseur dans une section d'une configuration. Reçoit une référence sur un objet Config (cf. module confmod) et le nom de la section. Les sous-classes de `DataConverter` qui ont des paramètres de fonctionnement à sauvegarder doivent redéfinir cette méthode, dans laquelle elles doivent commencer par appeler la méthode héritée de la classe parente, et ensuite stocker leurs propres paramètres de fonctionnement.

has_h2lconversion

`conv.has_h2lconversion () → booléen`

`def has_h2lconversion (self) :`

Retourne la valeur de l'indicateur du support de conversion de l'unité 'high' vers l'unité 'low'. Il existe des synonymes de cette méthode, définis pour des raison de sémantique dans certains contextes d'utilisation `has_u2dconversion` (pour les relations unité utilisateur → unité data des grandeurs), `has_e2mconversion` (pour les relations unité codeur → unité moteur des grandeurs axes), `has_p2rconversion` (pour les relations valeurs pseudo → valeurs réelles des groupes de grandeurs).

has_l2hconversion

`conv.has_l2hconversion () → booléen`

`def has_l2hconversion (self) :`

Retourne la valeur de l'indicateur du support de conversion de l'unité 'low' vers l'unité 'high'. Il existe des synonymes de cette méthode, définis pour des raison de sémantique dans certains contextes d'utilisation `has_d2uconversion` (pour les relations unité data → unité utilisateur des grandeurs), `has_m2econversion` (pour les relations unité moteur → unité codeur des grandeurs axes), `has_r2pconversion` (pour les relations valeurs réelles → valeurs pseudo des groupes de grandeurs).

infos

`conv.infos () → chaîne d'informations`

`def infos (self) :`

Retourne une chaîne de caractères décrivant le convertisseur. Cette méthode est principalement destinée à l'utilisateur, pour avoir une information en ligne (pour les opérations plus automatiques, les scripts ont disposent de méthodes qui leurs permettent d'accéder individuellement aux paramètres). Les sous-classes de `DataConverter` peuvent redéfinir la méthode `do_infos` afin de concaténer à la chaîne générée par la classe parente (dont elles doivent donc appeler la méthode infos héritée) leurs propres informations. Un exemple de chaîne retournée par infos:

```
Converter of class DataConvLinear:
    -usermodifiable (user can modify parameters): 0
    -alwaysduplicate (force values duplication): 0
    -hnotbemodified (forbid 'high' values modification): 1
    -lnotbemodified (forbid 'low' values modification): 1
    -ofsh2l (high to low offset): 12.6552
```

```
    -coefh2l (high to low coefficient): 2.45
```

do_infos

`self.do_infos () → chaîne d'informations`

`def do_infos (self) :`

Les sous-classes de `DataConverter` peuvent redéfinir cette méthode afin de concaténer leurs propres informations à la chaîne générée par la classe parente (dont elles doivent donc appeler la méthode `do_infos` héritée) . Une ligne d'information (hors la première ligne) est constituée d'une tabulation, d'un tiret, ensuite on a soit le nom de la méthode accesseur du paramètre et un petit descriptif entre parenthèses, soit juste un descriptif du paramètre (sans parenthèses) s'il n'y a pas d'accesseur, enfin un caractère deux-points et la valeur ; chaque ligne (y compris la dernière) se terminant par un caractère retour à la ligne (\n).

paramsdesc

`conv.paramsdesc () → liste de tuples de description`

`def paramsdesc (self) :`

Retourne une liste de tuples décrivant les paramètres modifiables du convertisseur. La liste est constituée de tuples contenant chacun : (access, name, type, description, valid).

access

> Le type d'accès au paramètre (string). Peut-être `"priv"` lorsqu'il s'agit d'un membre privé dont on veut éventuellement permettre la modification dans des dialogues de configuration. Doit être `"pub"` lorsqu'il s'agit d'un membre public (accessible par une méthode accesseur).

name

> Le nom du paramètre (string). Doit être le nom de la variable membre concernée s'il s'agit d'un membre privé (avec le préfixe "_NomClasse" si la variable a un nom commençant par deux underscore). Doit être le nom de la méthode accesseur s'il s'agit d'un membre public.

type

> Le type du paramètre (string). Peut prendre une valeur parmis : `"bool"`, `"int"`, `"float"`, `"string"`, `"eval"`. Le type "eval" indique que la valeur sous forme de chaine est une expression Python qui pourra être évaluée.

description

> Une description du paramètre (string). Peut comporter éventuellement plusieurs lignes. Doit expliquer à quoi sert le paramètre, quelles valeurs il peut prendre…

valid

> Validation du paramètre (fonction ou chaine).

S'il est fourni sous forme de fonction (incluant bounded-method, callable-object, lambda-expression), elle doit prendre deux paramètres : l'objet de conversion concerné, et la valeur à vérifier ; et retourner : `None` si la valeur est correcte pour le paramètre, et une chaine d'explication si la valeur est incorrecte.

S'il est fourni sous forme de chaine, celle-ci doit correspondre à une expression évaluable logiquement, contenant un paramètre *`x`* représentant la valeur pour le paramètre décrit par le tuple, et valant vrai si la valeur est correcte (si elle sort en exception, on considère qu'elle vaut faux). Cela permet d'écrire des expressions simples comme `"x>0"` sans avoir à définir une fonction pour cela. Lorsque cette expression est évaluée, le module `math` est rendu disponible (on peut appeler `math.sin(x)`…).

S'il est à `None`, aucune validation autre que le contrôle de type n'est effectuée.

Interface Utilisateur. La méthode `paramsdesc` est destinée principalement aux concepteurs d'interfaces utilisateurs (graphiques ou en ligne de commande), afin de leur permettre d'automatiser la saisie de ces paramètres. Il leur est demandé de ne jamais afficher les noms des variables membres privés données dans le tuple (uniquement les descriptions), et par contre d'afficher les noms des méthodes accesseurs pour les paramètres publiques (c'est normalement le seul nom connu dans la description d'une variable publique). De plus, pour l'évaluation des expression chaine de validation, il leur est demandé de rendre accessible le contenu du module math (avec le préfix `math.`…) lors de cette évaluation.

do_paramsdesc

`self.do_paramsdesc () → liste de tuples de description`

`def do_paramsdesc (self)`

Les sous-classes de `DataConverter` peuvent redéfinir cette méthode afin de concaténer à la liste générée par la classe parente (dont elles doivent donc appeler la méthode `do_paramsdesc` héritée) leurs propres tuples d'informations (voir la description de ce tuple dans DataConverter.paramsdesc).

hmanip

`conv.hmanip () → manipulateur de données "high"`

`def hmanip (self) :`

Retourne le manipulateur de données pour les valeurs exprimées en unité 'high'. Cet objet est une instance d'une sous-classe de `DataManipulator`, et il est stocké dans la variable membre `__hmanip`. Il existe des synonymes de cette méthode, définis pour des raison de sémantique dans certains contextes d'utilisation `umanio` (pour les relations unité utilisateur → unité data des grandeurs), `emanip` (pour les relations unité codeur → unité moteur des grandeurs axes), `pmanip` (pour les relations valeurs pseudo → valeurs réelles des groupes de grandeurs).

lmanip

`conv.lmanip () → manipulateur de données "low"`

`def lmanip (self) :`

Retourne le manipulateur de données pour les valeurs exprimées en unité 'low'. Cet objet est une instance d'une sous-classe de `DataManipulator`, et il est stocké dans la variable membre `__lmanip`. Il existe des synonymes de cette méthode, définis pour des raisons de sémantique dans certains contextes d'utilisation : `dmanip` (pour les relations unité data → unité utilisateur des grandeurs), `mmanip` (pour les relations unité moteur → unité codeur des grandeurs axes), `rmanip` (pour les relations valeurs réelles → valeurs pseudo des groupes de grandeurs).

DataConvLinear

Nom

`DataConvLinear` — Classe de conversion linéaire.

Synopsis

Module: `pyexp.DataConvLinearMod`

Classe parente: `DataConverter`

Cette classe applique une simple fonction de conversion linéaire sous la forme `y = a x + b`. Plus pécisément : `low = coefh2l * high + ofsh2l`.

Configuration

Section CONVERTER

Dans la section de configuration indiquée au loadconfig (par défaut CONVERTER), on pourra trouver - en plus des valeurs provenant de DataConverter - les noms de valeurs suivants :

coefh2l

> Valeur nombre décimal flottant représentant le coefficient de conversion pour passer de l'unité 'high' à l'unité 'low'. En cas d'absence dans la section de configuration la valeur 1.0 est utilisée. Cette valeur est stockée dans la variable membre __high2low_coef (en passant par la méthode coefh2l).

ofsh2l

> Valeur nombre décimal flottant représentant l'offset de conversion pour passer de l'unité 'high' à l'unité 'low'. En cas d'absence dans la section de configuration la valeur 0.0 est utilisée. Cette valeur est stockée dans la variable membre __high2low_ofs (en passant par la méthode ofsh2l).

Variables Membres

__high2low_coef

> Nombre flottant, coefficient de conversion lors du passage d'unité high vers unité low. Initialisé à 1.0 lors de la construction d'un `DataConvLinear`. Chargé lors de la configuration à partir de la valeur flottante stockée sous le nom `coefh2l` dans la configuration (voir section Configuration). Consultable et modifiable avec la méthode `coefh2l`. Valeur utilisée dans les méthodes de conversion.

__high2low_ofs

> Nombre flottant, offset de conversion lors du passage d'unité high vers unité low. Initialisé à 0.0 lors de la construction d'un `DataConvLinear`. Chargé lors de la configuration à partir de la valeur flottante stockée sous le nom `ofsh2l` dans la configuration (voir section Configuration). Consultable et modifiable avec la méthode `ofsh2l`. Valeur utilisée dans les méthodes de conversion.

__init__

`conv = DataConvLinear ()`

`def __init__ (self) :`

Constructeur d'un DataConvLinear. Appelle la méthode de construction héritée DataConverter.__init__, puis initialise les variables membres.

do_loadconfig

Redéfinit la méthode héritée DataConverter.do_loadconfig, en appelant cette méthode héritée puis en chargeant les données de configuration propres à `DataConvLinear`.

do_storeconfig

Redéfinit la méthode héritée DataConverter.do_storeconfig, en appelant cette méthode héritée puis en sauvegardant les données de configuration propres à `DataConvLinear`.

do_h2lconversion

Redéfinit la méthode héritée DataConverter.do_h2lconversion, en effectuant le calcul `résultat_low = valeur_high * coefh2l + ofsh2l`.

do_l2hconversion

Redéfinit la méthode héritée DataConverter.do_l2hconversion, en effectuant le calcul `résultat_high = (valeur_low - ofsh2l) / coefh2l`. Un contrôle est effectué sur le coefficient afin de contrôler qu'il n'est pas nul (s'il est nul, une exception `ZeroDivisionError` est levée).

do_infos

Redéfinit la méthode héritée DataConverter.do_infos, en appelant cette méthode héritée puis en y ajoutant les informations propres à `DataConvLinear`.

do_paramsdesc

Redéfinit la méthode héritée DataConverter.do_paramsdesc, en appelant cette méthode héritée puis en y ajoutant la description des paramètres propres à `DataConvLinear`.

coefh2l

`conv.coefh2l ([coefficient]) → coefficient`

`def coefh2l(self,coef = None) :`

Accès à - et modification du - coefficient de conversion d'unité high vers unité low. Si *`coef`* est à `None`, le coefficient stocké dans la variable membre `__high2low_coef` n'est pas modifié et sa valeur est simplement retournée. Si *`coef`* n'est pas à `None`, sa représentation flottante est stockée dans la variable membre `__high2low_coef`, dont l'ancienne valeur est retournée.

ofsh2l

`conv.ofs2l ([offset]) → offset`

`def ofsh2l(self,ofs = None) :`

Accès à - et modification de - l'offset de conversion d'unité high vers unité low. Si ***`ofs`*** est à `None`, l'offset stocké dans la variable membre `__high2low_ofs` n'est pas modifié et sa valeur est simplement retournée. Si ***`ofs`*** n'est pas à `None`, sa représentation flottante est stockée dans la variable membre `__high2low_ofs`, dont l'ancienne valeur est retournée.

DataConvNone

Nom

`DataConvNone` — Classe de "non" conversion (ou conversion nulle).

Synopsis

Module: `pyexp.DataConvNoneMod`

Classe parente: `DataConverter`

Cette classe n'implémente aucun calcul dans ses fonctions de conversion, celles-ci se contentent de retourner les valeurs qu'on leur donne. Elle n'utilise aucune information de configuration hors celles héritées de `DataConverter`.

Membres

Aucune variable membre supplémentaire..

do_h2lconversion

Redéfinit la méthode héritée DataConverter.do_h2lconversion, en se contentant de retourner la valeur donnée en paramètre.

do_l2hconversion

Redéfinit la méthode héritée DataConverter.do_l2hconversion, en se contentant de retourner la valeur donnée en paramètre.

DataManipulator

Nom

`DataManipulator` — Classe abstraite pour la hiérarchie des classes de manipulateurs de données.

Synopsis

Module: `pyexp.DataManipulatorMod`

Cette classe donne le cadre dans lequel les sous classes de manipulation doivent être définies afin d'être utilisables de façon transparente par pyexp. Elle fourni un support par le biais de méthodes `tool_…` permettant certaines opérations (lecture/écriture…), et offre des comportements par défaut dans certaines méthodes redéfinissables, ceci afin d'éviter d'avoir à écrire du code dans les sous-classes. Elle ne doit pas être instanciée directement, **seules des instances de sous-classes, liées à des types précis de données, doivent être créées**.

Options de Formatage

Des méthodes de DataManipulator permettent de traiter les entrées/sorties texte et binaire d'une façon standardisée. Pour offrir différentes possibilités de formatage, des options peuvent être fournies sous la forme de clées associées à des valeurs dans deux dictionnaires (un dictionnaire pour les options globales au fichier, et un dictionnaire pour les options spécifiques à la valeur). Si une clée d'option est absente mais nécessaire, une valeur par défaut est utilisée. Certaines valeurs d'options sont directement des chaînes de formatage à laquelles on applique l'opérateur % de Python, se référer à la documentation Python - 2.1.5.2 String Formatting Operations (http://www.python.org/doc/current/lib/typesseq-strings.html) pour plus de détails (et ne pas hésiter à ouvrir une console Python et à faire des tests), voir entre autre dans certains cas l'utilisation de `"%(nom)…"` associé à un dictionnaire contenant une clé `"nom"`.

Etat actuel. Les options décrites actuellement ont été choisies lors de l'écriture des méthodes `tool_…`, afin de supporter quelques options de formatage. Il sera sûrement nécessaires de les revoir à l'épreuve d'une utilisation large, mais ceci n'a normalement pas d'incidence sur le reste du code.

Futur. Pyexp doit de plus, à terme, être capable d'écrire dans un format de fichier scientifique standardisé comme netCDF (http://www.unidata.ucar.edu/packages/netcdf/) (network Common Data Form), ou encore HDF (http://hdf.ncsa.uiuc.edu/) (Hierarchical Data Format). Des modules d'interface Python vers les librairies de manipulation de ces formats de fichiers existent pour netCDF (http://www.unidata.ucar.edu/packages/netcdf/software.html#Python). Reste à voir à quel niveau il faut insérer du code pour manipuler de telles librairies.

Options Fichiers

Elle sont donc fournies par le biais d'un dictionnaire - généralement donné comme paramètre *`fileopt`* - dans lequel chaque clé correspond à une option (obligatoire ou optionnelle). Voici la liste des clés

file

Objet fichier (ou offrant la même interface) utilisé pour la lecture ou l'écriture.

binmode

Indicateur d'entrées/sorties en mode binaire. Si cet indicateur est présent - quelle que soit la valeur associée - les entrées/sorties se font au format binaire. S'il est absent - le défaut - les entrées/sorties se font au format texte.

encoding

Encodage des caractères écrits en mode texte. Les valeurs possibles sont celles utilisable pour le paramètre *`encoding`* de la méthode `encode` des chaînes Python. Par défaut on utilise simplement l'encodage des chaînes de caractère en mémoire (ie. on écrit directement les chaînes sans les reformater).

linesep

Séparateur de ligne lors de l'écriture de la représentation en mode texte. Par défaut on utilise le séparateur de ligne de la plateforme hôte tel que défini dans la variable Python `linesep` du module `os` (`"\r\n"` sur les PCs sous Windows, `"\n"` sur les machines sous Unix, et `"\r"` sur les Macintosh).

colsep

Séparateur de colonne lors de l'écriture de la représentation en mode texte. On entend par cela le séparateur entre les représentations textuelles de deux valeurs consécutives sur la même ligne. Par défaut on utilise le caractère tabulation "\t", rarement présent dans les chaînes de caractères, occupant peu de place, et assurant une relecture et une présentation correcte des fichiers dans la pluspart des programmes.

Options Valeurs

Elle sont donc fournies par le biais d'un dictionnaire - généralement donné comme paramètre *`valopt`* - dans lequel chaque clé correspond à une option (obligatoire ou optionnelle). Voici la liste des clés :

itxtformat

Représentation à utiliser lors de l'écriture de la représentation textuelle de nombres entiers. On se base sur l'opérateur % appliqué à la valeur à enregistrer. Par défaut on utilise simplement le format `"%d"`.

ibinformat

Représentation à utiliser lors de l'écriture de la représentation binaire de nombres entiers.

ftxtformat

Représentation à utiliser lors de l'écriture de la représentation textuelle de nombres flottants. On se base sur l'opérateur % appliqué à la valeur à enregistrer. Par défaut on utilise simplement le format `"%g"`.

fbinformat

Représentation à utiliser lors de l'écriture de la représentation binaire de nombres flottants.

stxtformat

Représentation à utiliser lors de l'écriture de la représentation textuelle de chaînes de caractères. On se base sur l'opérateur % appliqué à la valeur à enregistrer. Par défaut on utilise simplement le format `"%s"`.

sbinformat

Représentation à utiliser lors de l'écriture de la représentation binaire de chaînes de caractères.

valsep

Séparateur de valeurs de base lors de l'écriture de la représentation en mode texte de vecteurs. Par défaut on utilise le caractère indiqué dans l'option "colsep" pour le fichier (défaut "\t").

rawsep

Séparateur de vecteurs lignes lors de l'écriture de la représentation en mode texte de matrices. Par défaut on utilise le caractère indiqué dans l'option "linesep" pour le fichier (caractère retour à la ligne de la plateforme par défaut).

matsep

Séparateur de matrices lors de l'écriture de la représentation en mode texte de série de matrices. Par défaut on utilise le caractère indiqué dans l'option "linesep" pour le fichier (caractère retour à la ligne de la plateforme par défaut).

vecthead

En-tête de vecteurs lors de la représentation en mode texte de vecteurs (ou matrices). Ecrit juste avant la première valeur du vecteur. Si cette option est présente, elle doit être une chaine de formatage qui sera formatée en utilisant un dictionnaire contenant les clés suivantes :

vectnum

Numéro du vecteur. Dans le cas d'un vecteur isolé, toujours la valeur 0. Dans le cas d'un vecteur ligne d'une matrice, un nombre entier à partir de 1.

valcount

Nombre de valeurs dans le vecteur.

valsep

Séparateur de valeur courant (voir option de valeur `"valsep"`).

colsep

Séparateur de colonnes courant (voir option de fichier colsep).

vectqueue

Comme vecthead, mais écrit à la fin du vecteur (juste après sa dernière valeur).

mathead

En-tête de vecteurs lors de la représentation en mode texte de matrices. Ecrit juste avant la première ligne de la matrice. Si cette option est présente, elle doit être une chaine de formatage qui sera formatée en utilisant un dictionnaire contenant les clés suivantes :

matnum

Numéro de la matrice. Dans le cas d'une matrice isolée, toujours la valeur 0. Dans le cas d'une série de matrices, un nombre entier à partir de 1 représentant le numéro de la matrice dans la série.

rawcount

Nombre de lignes de la matrice.

colcount

Nombre de colonnes de la matrice.

valsep

Séparateur de valeur courant (voir option de valeur valsep).

rawsep

Séparateur de vecteur lignes (voir option de valeur rawsep).

colsep

Séparateur de colonnes courant (voir option de fichier colsep).

linesep

Séparateur de colonnes courant (voir option de fichier colsep).

matqueue

Comme mathead, mais écrit à la fin de la matrice (juste après son dernier vecteur ligne).

__init__

```
manip = datamanipulator (identificateur)
def __init__ (self) :
```

makecopy

```
manip.makecopy (valeur) → valeur
def makecopy (self,value) :
```

Demande au manipulateur de créer une copie de la variable donnée en argument. La méthode appelle `do_makecopy`, qui peut être redéfini dans les sous-classes afin de faire la duplication suivant le type de donnée supporté par la classe effective de manipulateur.

do_makecopy

```
self.do_makecopy (valeur) → valeur
def do_makecopy (self,value) :
```

Réalise effectivement une copie de la valeur passée en argument et la retourne. Cette méthode peut être redéfinie dans les sous-classes afin de s'adapter aux types de données supportés. Si la méthode `do_makecopy` de `DataManipulator` est appelée, elle procède par défaut en utilisant la fonction `deepcopy` du module standard `copy` de Python (permet de supporter les copies des types de base ainsi que des séquences, même imbriquées, mais pas optimal par rapport à du code connaissant a priori le type de données manipulées).

checkaccept

`manip.checkaccept (valeur) → booléen`

`def checkaccept (self,value) :`

Vérifie que la valeur donnée en paramètre est bien compatible avec le manipulateur de données. Cette méthode appelle do_checkaccept qui se charge d'effectuer le contrôle. Retourne 1 si oui, et 0 si non.

do_checkaccept

`self.do_checkaccept (valeur) → booléen`

`def do_checkaccept (self,value) :`

Réalise effectivement le contrôle que la valeur donnée en paramètre est compatible avec le manipulateur de données. Retourne 1 si oui, et 0 si non. La méthode définie dans `DataManipulator` lève une exception `NotImplementedError`.

getdimsizes

`manip.getdimsizes (valeur) → dimensions (tuple de)`

`def getdimsizes (self,value) :`

Retourne une indication du nombre de dimensions de la valeur, et de la taille dans chaque dimension. Le résultat retourné est un tuple contenant les tailles dans chaque dimension. Pour une valeur simple, le résultat est un tuple vide (ex. `()`). Pour un vecteur à une dimension, le résultat est un tuple d'un seul élément contenant la taille du vecteur (ex. `(100,)`). Pour une matrice à deux dimensions (par exemple une image), le résultat est un tuple contenant la taille de la matrice dans les deux dimensions (ex. `(100,300)`). Une taille de valeur `-1` peut être indiquée afin de spécifier une taille inconnue. L'utilisation d'un tuple permet de connaître le nombre de dimensions avec la fonction Python `len`. Cette méthode appelle do_getdimsizes qui se charge de retourner la liste des dimensions.

do_getdimsizes

`self.do_getdimsizes (valeur) → dimensions (tuple de)`

`def do_getdimsizes (self,value) :`

Méthode appelée automatiquement par getdimsizes Réalise effectivement le calcul du nombre de dimensions et de la taille de chaque dimension de la valeur (voir description du fonctionnement normal dans `getdimsizes`).

getbasemanipulator

`manip.getbasemanipulator (valeur) → manipulateur de données`

`def getbasemanipulator (self,value) :`

Retourne un manipulateur de données pour un élément de base de la valeur. Par exemple, pour une matrice 2D d'entiers, retourne un manipulateur pour des entiers. Cette méthode appelle `do_getbasemanipulator` qui se charge de retourner le manipulateur de données de base.

do_getbasemanipulator

`self.do_getbasemanipulator (valeur) → manipulateur de données`

`def do_getbasemanipulator (self,value) :`

Méthode appelée automatiquement par `getbasemanipulator`. La méthode récupère le type de données retourné par `getbasetype`, et s'il s'agit d'un type de base int, long ou float, demande à la fonction datamanipulator de retourner un manipulateur correspondant, sinon une exception est levée.

getbasetype

`manip.getbasetype (valeur) → type de données`

`def getbasetype (self,value) :`

Retourne le type pour un élement de base de la valeur. Par exemple, pour une matrice 2D d'entiers, retourne un type Python `<type 'int'>`. Cette méthode appelle `do_getbasetype` qui se charge de retourner le type de base.

do_getbasetype

`self.do_getbasetype (valeur) → type de données`

`def do_getbasetype (self,value) :`

Méthode appelée automatiquement par `getbasetype`. La méthode par défaut se base sur les dimensions de la valeur telles que retournées par `getdimsizes`, et accède à une valeur de base en utilisant les indexs 0, elle retourne finalement le type Python de cette valeur de base.

compare

`manip.compare(valeur1, valeur2) → constante de comparaison`

`def compare (self,v1,v2) :`

Compare deux valeurs. Retourne une des constantes parmis : `COMP_V1EQV2`, `COMP_V1GTV2`, `COMP_V1LTV2`,

Description des constantes retournées :

COMP_V1EQV2

Valeur 0. Indique l'égalité entre ***`valeur1`*** et ***`valeur2`***.

COMP_V1GTV2

Valeur 1. Indique que ***`valeur1`*** est plus grand que ***`valeur2`***.

COMP_V1LTV2

Valeur -1. Indique que *`valeur1`* est plus petit que *`valeur2`*.

Cette méthode appelle `do_compare` qui se charge d'effectuer la sommation.

do_compare

`self.do_compare(valeur1, valeur2)` → `constante de comparaison`

```
def do_compare (self,v1,v2) :
```

Méthode appelée automatiquement par `compare`. Réalise effectivement la comparaison de *`v1`* et *`v2`*. La méthode par défaut utilise simplement les opérateurs de comparaison `<`, `>` et `==` de Python et retourne la constante de comparaison adéquate.

add

`manip.add (valeur1, valeur2)` → `somme des valeurs`

```
def add (self,v1,v2) :
```

Ajoute v2 à v1 et retourne le résultat. Cette méthode appelle `do_add` qui se charge d'effectuer la sommation.

do_add

`manip.do_add (valeur1, valeur2)` → `somme des valeurs`

```
def do_add (self,v1,v2) :
```

Méthode appelée automatiquement par `add`. Réalise effectivement la sommation de *`v1`* et *`v2`*. La méthode par défaut utilise simplement l'opérateur `+` de Python appliqué à `v1 + v2`.

subst

`manip.subst (valeur1, valeur2)` → `soustraction des valeurs`

```
def subst (self,v1,v2) :
```

Retire v2 à v1 et retourne le résultat. Cette méthode appelle `do_subst` qui se charge d'effectuer la soustraction.

do_subst

`manip.add (valeur1, valeur2)` → `soustraction des valeurs`

```
def add (self,v1,v2) :
```

Méthode appelée automatiquement par `subst`. Réalise effectivement la soustraction de *`v2`* de *`v1`*. La méthode par défaut utilise simplement l'opérateur `-` de Python appliqué à `v1 - v2`.

getstreamrepr

`manip.getstreamrepr (valeur)` → `chaine de données représentant la valeur`

`def getstreamrepr (self,value,fileopt=None,valopt=None) :`

Cette méthode redirige l'écriture de la valeur avec la méthode writestream, en la faisant se réaliser dans une chaîne de caractères qui est retournée (via la classe Python `StringIO`). Voir les options de formatage.

writestream

`manip.writestream (valeur, options fichier) [→ nombre d'octets écrits]`

`def writestream (self,value,fileopt,valopt=None) :`

Ecrit la valeur dans un fichier (indiqué via la clé `"file"` dans le dictionnaire d'options ***`fileopt`***). Voir les options de formatage. Si le flot d'écriture supporte l'opération `tell`, la méthode retourne le nombre d'octets écrits, sinon elle retourne `None`. Suivant le type de flot indiqué par la présence ou non de l'option de fichier `"binmode"`, la méthode appelle `do_writebinstream` ou `do_writestrstream` qui se chargent d'effectuer l'écriture avec le bon formatage.

do_writebinstream

`self.do_writebinstream (valeur, options fichier, options valeur)`

`def do_writebinstream (self,value,fileopt,valopt) :`

Réalise effectivement l'écriture de la valeur au format binaire dans un fichier. La méthode par défaut se base sur le type des valeurs de base tel que retourné par `getbasetype`, et sur la dimension des données tel que retourné par `getdimsizes`, et elle appelle les outils `tool_…` correspondant au support pour les types entier/entier long, flottant, et chaîne, dans les dimensions valeur simple, vecteur et matrice 2D. Pour les autres types de données et les dimensions supérieures, la méthode lève une exception `NotImplementedError`. Il est possible, dans une sous classe, de se baser sur le comportement par défaut ; mais pour des raisons d'optimisation il est préférable de redéfinir cette méthode et d'y placer directement un appel à la méthode `tool_…` adéquate, ou encore carrément le code d'écriture si les fonctions outils ne suffisent pas.

do_writestrstream

`self.do_writestrstream (valeur, options fichier, options valeur)`

`def do_writestrstream (self,value,fileopt,valopt) :`

Réalise effectivement l'écriture de la valeur au format texte dans un fichier. La méthode par défaut se base sur le type des valeurs de base tel que retourné par `getbasetype`, et sur la dimension des données tel que retourné par `getdimsizes`, et elle appelle les outils `tool_…` correspondant au support pour les types entier/entier long, flottant, et chaîne, dans les dimensions valeur simple, vecteur et matrice 2D. Pour les autres types de données et les dimensions supérieures, la méthode lève une exception `NotImplementedError`. Il est possible, dans une sous classe, de se baser sur le comportement par défaut ; mais pour des raisons d'optimisation il est préférable de redéfinir cette méthode et d'y placer directement un appel à la méthode `tool_…` adéquate, ou encore carrément le code d'écriture si les fonctions outils ne suffisent pas.

readstream

`manip.readstream (options fichier, options valeur) → valeur`

`def readstream (self,fileopt,valopt=None) :`

Lit la valeur à partir d'un fichier (indiqué via la clé `"file"` dans le dictionnaire d'options ***`fileopt`***). Voir les options de formatage. Suivant le type de flot indiqué par la présence ou non de l'option de fichier `"binmode"`, la méthode appelle `do_readbinstream` ou `do_readstrstream` qui se chargent d'effectuer la lecture.

do_readbinstream

`self.do_readbinstream (fichier, options fichier, options valeur) → valeur`

`def do_readbinstream (self,fileopt,valopt) :`

Réalise effectivement la lecture de la valeur au format binaire dans un fichier. Cette méthode n'offre actuellement aucune implémentation par défaut, et lève simplement une exception `NotImplementedError`.

do_readstrstream

`self.do_readstrstream (fichier, options fichier, options valeur) → valeur`

`def do_readstrstream (self,fileopt,valopt) :`

Réalise effectivement la lecture de la valeur au format texte dans un fichier. Cette méthode n'offre actuellement aucune implémentation par défaut, et lève simplement une exception `NotImplementedError`.

infos

`manip.infos () → chaîne d'informations`

`def infos (self) :`

Retourne une chaîne de caractères décrivant le manipulateur. Les sous-classes de `DataManipulator` peuvent redéfinir la méthode `do_infos` afin de concaténer à la chaîne générée par la classe parente (dont elles doivent donc appeler la méthode infos héritée) leurs propres informations. Un exemple de chaîne retournée par infos:

```
Manipulator of class DataManipInt:
    -support of basic integer types.
    -binary data size: 4 bytes for int, variable for long
```

```
    -number of dimensions: 0
```

do_infos

`self.do_infos () → chaîne d'informations`

`def do_infos (self) :`

Les sous-classes de `DataManipulator` peuvent redéfinir cette méthode afin de concaténer leurs propres informations à la chaîne générée par la classe parente (dont elles doivent donc appeler la méthode `do_infos` héritée) . Une ligne d'information (hors la première ligne) est constituée d'une tabulation, d'un tiret, puis d'informations terminées par un retour à la ligne (\n).

tool_writebinstream_int

`def tool_writebinstream_int (self,value,fileopt,valopt) :`

Méthode outil pour écrire une simple valeur entière (ou longue) au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writebinstream_int1D

`def tool_writebinstream_int1D (self,value,fileopt,valopt) :`

Méthode outil pour écrire un vecteur de valeurs entières (ou longues) au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writebinstream_int2D

`def tool_writebinstream_int2D (self,value,fileopt,valopt) :`

Méthode outil pour écrire une matrice 2D de valeurs entières (ou longues) au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writestrstream_int

`def tool_writestrstream_int (self,value,fileopt,valopt) :`

Méthode outil pour écrire une simple valeur entière (ou longue) au format texte. Voir tool_writestrstream.

tool_writestrstream_int1D

`def tool_writestrstream_int1D (self,value,fileopt,valopt,vectsize=None) :`

Méthode outil pour écrire un vecteurs de valeurs entières ou longues au format texte. Voir tool_writestrstream_1D.

tool_writestrstream_int2D

`def tool_writestrstream_int2D (self,value,fileopt,valopt,sizes=None) :`

Méthode outil pour écrire une matrice de valeurs entières ou longues au format texte. Voir tool_writestrstream_2D.

tool_writebinstream_float

```
def tool_writebinstream_int (self,value,fileopt,valopt) :
```

Méthode outil pour écrire une simple valeur flottante au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writebinstream_float1D

```
def tool_writebinstream_int1D (self,value,fileopt,valopt) :
```

Méthode outil pour écrire un vecteur de valeurs flottantes au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writebinstream_float2D

```
def tool_writebinstream_int2D (self,value,fileopt,valopt) :
```

Méthode outil pour écrire une matrice 2D de valeurs flottantes au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writestrstream_float

```
def tool_writestrstream_float (self,value,fileopt,valopt) :
```

Méthode outil pour écrire une simple valeur flottante au format texte. Voir tool_writestrstream.

tool_writestrstream_float1D

```
def tool_writestrstream_float1D
(self,value,fileopt,valopt,vectsize=None) :
```

Méthode outil pour écrire un vecteurs de valeurs flottanttes au format texte. Voir tool_writestrstream_1D.

tool_writestrstream_float2D

```
def tool_writestrstream_float2D
(self,value,fileopt,valopt,sizes=None) :
```

Méthode outil pour écrire une matrice de valeurs entières ou longues au format texte. Voir tool_writestrstream_2D.

tool_writebinstream_str

```
def tool_writebinstream_int (self,value,fileopt,valopt) :
```

Méthode outil pour écrire une simple valeur chaîne au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writebinstream_str1D

```
def tool_writebinstream_int1D (self,value,fileopt,valopt) :
```

Méthode outil pour écrire un vecteur de valeurs chaînes au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writebinstream_strt2D

```
def tool_writebinstream_int2D (self,value,fileopt,valopt) :
```

Méthode outil pour écrire une matrice 2D de valeurs chaînes au format binaire. Cette méthode n'est pas encore écrite (travail à terminer), et lève une exception NotImplementedError.

tool_writestrstream_str

```
def tool_writestrstream_str (self,value,fileopt,valopt) :
```

Méthode outil pour écrire une simple valeur chaîne au format texte. Voir tool_writestrstream.

tool_writestrstream_str1D

```
def tool_writestrstream_str1D
(self,value,fileopt,valopt,vectsize=None) :
```

Méthode outil pour écrire un vecteurs de chaines au format texte. Voir tool_writestrstream_1D.

tool_writestrstream_str2D

```
def tool_writestrstream_str2D (self,value,fileopt,valopt,sizes=None)
:
```

Méthode outil pour écrire une matrice de valeurs entières ou longues au format texte. Voir tool_writestrstream_2D.

tool_writestrstream

```
def tool_writestrstream (self,value,fileopt,valopt,format) :
```

Méthode outil pour écrire une simple valeur à laquelle on applique un formatage. La valeur à écrire est passée dans le paramètre ***`value`***, les dictionnaires d'options dans les paramètres ***`fileopt`*** et ***`valopt`***, et la chaine de formatage dans le paramètre ***`format`***.

tool_writestrstream_1D

```
def tool_writestrstream_1D
(self,value,fileopt,valopt,format,vectsize=None) :
```

Méthode outil pour écrire un vecteur de valeurs simples auxquelles on applique un formatage. Le vecteur de valeurs à écrire est passée dans le paramètre ***`value`***, les dictionnaires d'options dans les paramètres ***`fileopt`*** et ***`valopt`***, et la chaine de formatage dans le paramètre ***`format`***, on donne aussi optionnellement le nombre de valeurs du vecteur dans le paramètre ***`vectsize`***.

tool_writestrstream_2D

```
def                                                tool_writestrstream_2D
(self,value,fileopt,valopt,format,sizes=None) :
```

Méthode outil pour écrire une matrice 2D de valeurs simples auxquelles on applique un formatage. La matrice de valeurs à écrire est passée dans le paramètre ***`value`***, les dictionnaires d'options dans les paramètres ***`fileopt`*** et ***`valopt`***, et la chaine de formatage dans le paramètre ***`format`***, on donne aussi optionnellement les dimensions de la matrice dans le paramètre ***`sizes`*** (tel que retourné par la méthode getdimsizes).

DataManipInt

Nom

`DataManipInt` — Classe de manipulation de données entiers ou entiers longs.

Synopsis

Module: `pyexp.DataManipIntMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type entier ou entier long. Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfini la méthode héritée DataManipulator.do_makecopy, en retournant simplement la valeur.

do_checkaccept

Redéfini la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien de type entier ou entier long Python.

do_getdimsizes

Redéfini la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple vide (indicateur d'une donnée de base sans dimension).

do_getbasemanipulator

Redéfini la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les entiers (ie. un manipulateur de la même classe).

do_getbasetype

Redéfini la méthode héritée DataManipulator.do_getbasetype, retourne le type entier.

do_writebinstream

Redéfini la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_int.

do_writestrstream

Redéfini la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_int.

do_infos

Redéfini la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipInt1D

Nom

`DataManipInt1D` — Classe de manipulation de données tableaux d'entiers ou d'entiers longs.

Synopsis

Module: `pyexp.DataManipInt1DMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type tableau d'entier ou tableau d'entiers longs, sous la forme de tuples ou de listes. Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant une copie de la séquence (utilisation de `[:]` sur la valeur).

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien un tuple ou une liste, n'est pas vide, et en vérifiant que la valeur de base à l'indice 0 est bien de type entier ou entier long (on ne vérifie pas toute la séquence).

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple ne contenant qu'un item : la taille de la séquence.

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les entiers.

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, retourne le type entier.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_int1D.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_int1D.

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipInt2D

Nom

`DataManipInt2D` — Classe de manipulation de données tableaux à 2 dimensions d'entiers ou d'entiers longs.

Synopsis

Module: `pyexp.DataManipInt2DMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type tableau à 2 dimensions d'entier ou d'entiers longs, sous la forme de tuples ou de listes (de tuples ou de listes). Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant une copie de la séquence de séquences (utilisation de `[:]` sur les sous-séquences, et de `append` - transformation en tuple si la valeur d'origine est un tuple).

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien un tuple ou une liste non vide, que la valeur à l'indice 0 est bien un tuple ou une liste non vide, et en vérifiant que la valeur de base à l'indice 0,0 est bien de type entier ou entier long (on ne vérifie pas toute la séquence).

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple contenant deux un items, la taille de la séquence au premier niveau, et la taille des séquences de second niveau (si elles ne sont pas de même taille, utilisation de la valeur -1).

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les entiers.

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, retourne le type entier.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_int1D.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_int1D.

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipFloat

Nom

`DataManipFloat` — Classe de manipulation de données flottantes.

Synopsis

Module: `pyexp.DataManipFloatMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type flottant (équivalent aux `double` du langage C). Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant simplement la valeur.

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien de type flottant Python.

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple vide (indicateur d'une donnée de base sans dimension).

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les flottants (ie. un manipulateur de la même classe).

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, en retournant simplement le type Python pour les flottants.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_float.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_float.

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipFloat1D

Nom

`DataManipFloat1D` — Classe de manipulation de données tableaux de flottants.

Synopsis

Module: `pyexp.DataManipFloat1DMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type tableau de flottants, sous la forme de tuples ou de listes. Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant une copie de la séquence (utilisation de `[:]` sur la valeur).

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien un tuple ou une liste, n'est pas vide, et en vérifiant que la valeur de base à l'indice 0 est bien de type flottant (on ne vérifie pas toute la séquence).

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple ne contenant qu'un item : la taille de la séquence.

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les flottants.

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, retourne le type flottant.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_float1D.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_float1D.

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipFloat2D

Nom

`DataManipFloat2D` — Classe de manipulation de données tableaux à 2 dimensions de flottants.

Synopsis

Module: `pyexp.DataManipFloat2DMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type tableau à 2 dimensions de flottants, sous la forme de tuples ou de listes (de tuples ou de listes). Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant une copie de la séquence de séquences (utilisation de `[:]` sur les sous-séquences, et de `append` - transformation en tuple si la valeur d'origine est un tuple).

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien un tuple ou une liste non vide, que la valeur à l'indice 0 est bien un tuple ou une liste non vide, et en vérifiant que la valeur de base à l'indice 0,0 est bien de type flottant (on ne vérifie pas toute la séquence).

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple contenant deux un items, la taille de la séquence au premier niveau, et la taille des séquences de second niveau (si elles ne sont pas de même taille, utilisation de la valeur -1).

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les flottants.

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, retourne le type flottant.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_float2D.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil .

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipStr

Nom

`DataManipStr` — Classe de manipulation de données chaînes (standard ou unicode).

Synopsis

Module: `pyexp.DataManipStrMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type chaîne (standard ou Unicode). Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant simplement la valeur (les chaînes sont immutables).

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien de type chaine standard ou chaîne Unicode.

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple vide (indicateur d'une donnée de base sans dimension).

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les chaînes (ie. un manipulateur de la même classe).

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, en retournant le type chaîne standard Python.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_float.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_float.

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipStr1D

Nom

`DataManipStr1D` — Classe de manipulation de données tableaux de chaînes (standard ou unicode).

Synopsis

Module: `pyexp.DataManipStr1DMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type tableau de chaînes (standard ou Unicode), sous la forme de tuples ou de listes. Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant une copie de la séquence (utilisation de `[:]` sur la valeur).

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien un tuple ou une liste, n'est pas vide, et en vérifiant que la valeur de base à l'indice 0 est bien de type chaîne standard ou chaîne Unicode (on ne vérifie pas toute la séquence).

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple ne contenant qu'un item : la taille de la séquence.

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les chaînes.

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, retourne le type chaîne standard Python.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_str1D.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_str1D.

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

DataManipStr2D

Nom

`DataManipStr2D` — Classe de manipulation de données tableaux à 2 dimensions de chaînes (standard ou unicode).

Synopsis

Module: `pyexp.DataManipStr2DMod`

Classe parente: `DataManipulator`

Cette classe implémente les méthodes de manipulation pour des valeurs Python de type tableau à 2 dimensions de chaînes (standard ou Unicode), sous la forme de tuples ou de listes (de tuples ou de listes). Elle hérite de la classe `DataManipulator` dont elle redéfinit certaines méthodes `do_`....

do_makecopy

Redéfinit la méthode héritée DataManipulator.do_makecopy, en retournant une copie de la séquence de séquences (utilisation de `[:]` sur les sous-séquences, et de `append` - transformation en tuple si la valeur d'origine est un tuple).

do_checkaccept

Redéfinit la méthode héritée ref-DataManipulator.do_checkaccept, en vérifiant que la valeur est bien un tuple ou une liste non vide, que la valeur à l'indice 0 est bien un tuple ou une liste non vide, et en vérifiant que la valeur de base à l'indice 0,0 est bien de type chaîne standard ou chaîne Unicode (on ne vérifie pas toute la séquence).

do_getdimsizes

Redéfinit la méthode héritée DataManipulator.do_getdimsizes, en retournant un tuple contenant deux un items, la taille de la séquence au premier niveau, et la taille des séquences de second niveau (si elles ne sont pas de même taille, utilisation de la valeur -1).

do_getbasemanipulator

Redéfinit la méthode héritée DataManipulator.do_getbasemanipulator, en retournant un manipulateur pour les chaines.

do_getbasetype

Redéfinit la méthode héritée DataManipulator.do_getbasetype, retourne le type chaîne standard Python.

do_writebinstream

Redéfinit la méthode héritée DataManipulator.do_writebinstream, en appelant la méthode outil DataManipulator.tool_writebinstream_str2D.

do_writestrstream

Redéfinit la méthode héritée DataManipulator.do_writestrstream, en appelant la méthode outil DataManipulator.tool_writestrstream_str2D.

do_infos

Redéfinit la méthode héritée DataManipulator.do_infos, en ajoutant ses propres informations à celles retournées par la méthode héritée.

PyExp - Abîmes

Laurent Pointal
Informaticien, développeur d'applications
C.N.R.S.

PyExp - Abîmes
par Laurent Pointal

Afin d'offrir des vues relativement simples aux développeurs de scripts expérimentaux, de classes de devices ou de groupes pour gérer des équipements, PyExp a un fonctionnement interne basé sur des règles simples. Mais l'application systématique de ces règles, récursivement sur différents objets, peut compliquer nettement les choses. Ce livre décrit le fonctionnement interne de PyExp, la façon dont les objets coopèrent et dont les fonctions de haut niveau sont traduites en termes de manipulations d'objets.