Recommandations à destination des développeurs

Table des matières

1. Introduction

Qu'est-ce que c'est ?

Ce wiki présente des recommandations à l'attention des développeurs de Pilot Systems. Ce ne sont pas des règles fixes. On peut les contourner. Mais il faut avoir des bonnes raisons pour.

Mises à jour

Ce document n'est pas à usage unique. Ne pas hésiter à poser des commentaires pour provoquer des discussions sur des règles qui ne paraissent pas pertinentes.

Auteurs et Licence

2. Commenter son code

L'homme est un animal qui commente. (Aristote)

Trois bonnes raisons de commenter son code

1. On ne travaille pas seul, le collègue qui doit reprendre le code est content d'avoir des commentaires.
2. Quand on reprend on projet après ne pas avoir touché au code pendant des mois, même si on l'a écrit, c'est bien d'avoir des commentaires.
3. Commenter le code (par exemple ce qu'une fonction prend en entrée et en sortie) aide à mieux comprendre où on va et à mieux structurer son code.

Commenter du code Python

Il existe deux manières de commenter du code Python.

Les docstrings

Les docstrings servent à commenter de manière globale une fonction, une classe ou un module. Les docstrings doivent expliquer ce que fait la fonction (où la classe), avec quels paramètres ont doit l'appeler, et ce qu'elle renvoie.

Dans un code professionnel, toute fonction, toute classe, toute méthode doit avoir une docstring.

On écrit la docstring dés le début, peut être sans tout préciser mais en donnant au moins l'intention de la fonction.

Les commentaires avec

Les commentaires avec # sont eux utilisés pour expliquer le fonctionnement interne du code, un choix d'architecture, ... Il faut les utiliser dans les cas suivants (liste non exhaustive) :

- algorithme un peu complexe, qui ne se comprend pas bien à la lecture ;
- "hack" dont la raison d'être ou l'utilité n'est pas claire au premier abord ;
- subdiviser une fonction en plusieurs parties logiques (lorsque la création de plusieurs fonctions n'est pas souhaitable) ;
- regrouper par catégorie les fonctions dans une classe, lorsqu'une classe possède beaucoup de fonctions ;
- ...

Commenter du code TAL

Parce que séparer la donnée de la mise en forme n'est pas toujours très facile et encore moins explicite, il est fortement encouragé de créer des blocs logiques, pour les traitements spécialisés ou les boucles, et de les nommer explicitement -> tal:monBloc

Pour les commentaires, particulièrement les très verbeux, un tal:comments contenant le texte et masqué du rendu par un replace="nothing" aidera la reprise aisée du code.

On peut faire de très beau commentaire en TAL :

```
<tal:comment tal:replace="nothing">
  FIX ME: attention, ça casse si X vaut 2. A corriger.
</tal:comment>
```

On peut même documenter des macros :

```
<tal:doc>
Name: display-blobs
Description: display blobs in a table
Parameters: blobs (a list of Blob objects)
</tal:doc>
<tal:display-blobs metal:define-macro="display-blobs">
  ...
</tal:display-blobs>
```

Divers

- ne jamais mettre du code en commentaire sans inclure une explication ("debug", "au cas où le client change d'avis et souhaite qu'on affiche les blobs", etc.). Un code en commentaire n'est pas pris en compte par l'application. S'il n'a pas été retiré du code, c'est qu'il y a une raison. Et on veut la connaître. Et dites aussi qui vous êtes.
- ne pas hésiter à mettre un FIXME en cas de doute mais, dans ce cas, l'expliquer (dire ce qu'il faudrait faire, pourquoi vous pensez qu'il faudrait corriger le code et que vous ne l'avez pas fait, etc.). Indiquer qui vous êtes ("FIXME: it will not work if X == 2 (DB)")

3. Documenter pour capitaliser

Quand on a eu du mal à trouver quelque chose ou que l'on tombe sur une chose intéressante il est bien de le documenter.

Donc :

- quand on a un problème :
  - ♦ toujours penser à regarder sur notre wiki interne et utiliser la merveilleuse fonctionnalité de recherche
  - ♦ s'il n'est pas abordé, on soumet à l'équipe
  - ♦ si c'est un minimum intéressant on documente une fois trouvé le pourquoi du parceque
- quand on documente :
  - ♦ c'est en général concis : on aborde un sujet précis par page
  - ♦ on met un max de mots significatifs (c'est pour cela que l'on fait des phrases) de manière à ce qu'une recherche puisse être fructueuse
  - ♦ on met des liens vers des ressources web pour "aller plus loin"
- penser au site web : pourquoi ne pas faire une entrée de blog avec ce que l'on vient de documenter ? Dans ce cas faire relire l'orthographe et attendre aussi les réactions de l'équipe pour être sur que l'on dit des choses justes.

Documenter permet aussi aux abonnés (et tout développeur de Pilot doit être abonné à ce wiki) de pouvoir réagir sur la solution apporté, soit en proposant plus simple ou plus souple soit en laçant un vieux troll ;-)

À noter aussi que l'on doit profiter des nouveaux arrivants pour avoir des pages de documentation sur les aspects basiques de la programmation Plone / Django etc... (mode tutorial).

4. Écrire du code propre

Motivation

Cette page contient quelques règles simples qui permettent d'écrire du code propre. Elles sont indépendantes du langage utilisé, même si le Python est bien sûr le plus présent dans nos esprits à la rédaction du document.

Les règles sont listées avec leur(s) justification(s), elles ne sont pas là uniquement pour embêter les gens. Pas uniquement, j'ai bien dit.

Mise en forme

Identation

Règles

Un code propre doit être indenté, avec une indentation homogène dans tout le code.

Seuls des espaces doivent être utilisés dans l'identation.. Bien configuré son éditeur à cet effet, il n'y a rien de pire que de méler tabulations et espaces.

Justifications

L'indentation est nécessaire pour rendre le code lisible (et impérative en Python).

L'indentation ne doit pas varier entre les éditeurs utilisés, le fichier ouvert, la présence ou non de préfixes (ex: numéro de lignes quand on grep) sur les lignes, ... d'où l'utilisation d'espaces, et non de tabulations.

Lignes de code

Règles

Les lignes de code ne doivent, sauf exceptions, pas dépasser 80 colonnes.

Justifications

1. C'est la taille standard d'un terminal.
2. Une ligne trop longue est fatiguante à suivre pour l'oeil.
3. Une taille standard permet de configurer son espace de travail afin d'avoir de manière optimale un ou plusieurs terminaux et un ou plusieurs fichiers textes ouverts sur l'écran sans retour à la ligne ou chevauchement.
4. Une ligne trop longue implique soit un niveau d'indentation trop profond (faire des sous fonctions) soit une ligne trop complexe (la scinder en deux).

Note

Python 3.0 ne reconnaitra plus l'antislash `\` final pour continuer la ligne sur la suivante, par contre l'interpréteur sait que si un `(`, '[' ou `{` est ouvert la ligne ne sera pas terminé tant qu'ils ne sont pas fermés.

Donc on écrit pas :

```
concatenated = "Que cette chaine est longue surement trop longue" \
       + "surtout si je dois en ajouter une sur une autre ligne etc..."
```

Mais plutôt :

```
concatenated = ( "Que cette chaine est longue surement trop longue"
       + "surtout si je dois en ajouter une sur une autre ligne etc..." )
```

Nomenclature des variables

On code en anglais

Règle

Le nom des fonctions, variables, méthodes, classes et les commentaires doivent être en anglais.

Justifications

1. On peut être amené à travailler en collaboration avec des développeurs non francophones.
2. On souhaite pouvoir diffuser le code sous une licence libre au près de la communauté.
3. Les identifiants du langage et des bibliothèques utilisées sont en anglais, coder en français implique un mélange peu esthétique des deux langues.

Clair ou court ?

Règle

Plus une fonction/variable/... est à usage externe, et plus son nom doit être explicite. Une variable locale dans une "liste compréhension" peut s'appeler `k` ou `l`, un paramètre de fonction ou le nom d'une fonction doit avoir un nom explicite.

Justifications

Il s'agit d'un compris entre la concision du code (qui permet de taper plus vite) et la volonté de savoir à quoi sert une variable (ou une fonction) facilement.

Structuration du code

Éviter les fonctions trop longues

Règle

Considérer que la taille maximale d'une fonction, suivant son rôle et sa complexité, est de 30 à 40 lignes. Au-delà, découper la fonction en plusieurs sous-fonctions.

Justifications

1. Une fonction trop longue est difficile à lire/comprendre.

2. Une fonction trop longue est difficile à surcharger par héritage ou à monkey-patcher.
3. Une fonction trop longue empêche la réutilisabilité du code, si une partie de son traitement peut être utile pour une autre fonction.
4. Une fonction trop longue rend plus difficile l'utilisation d'un debugguer comme `pdb` ou `gdb`.

Éviter les copier-coller

Règle

Avant de copier-coller du code, on se demande toujours au moins **3 fois** (et on demande à l'équipe) si on ne pourrait pas factoriser le code à la place.

Justifications

1. Il est facile de se tromper et de créer un bug en oubliant de modifier une partie du copier-coller.
2. Du code copier-coller est plus difficile à lire, car il faut manuellement voir les différences entre les deux morceaux de code.
3. Du code copier-coller est très difficile à maintenir (correction de bugs ou évolutions), car il faut modifier toutes les copies.

5. Dans Python, Zope et Plone

5.1. Squelettes de code

Esprit

Un bon développeur est un développeur feignant. Plus exactement, c'est un développeur qui sait réutiliser du code existant, connu pour fonctionner. Savoir écrire 100 lignes de code standard en 1h, c'est bien. Savoir copier-coller du code standard en 2 minutes, et personnaliser en 10 minutes, c'est mieux.

ZopeSkel

Note : l'utilisation de ZopeSkel est complètement facultative. Ne conserver *in-fine* que les parties intéressantes du produit (on ne fait un *egg* que si l'utilité est validée avec l'équipe).

ZopeSkel est le produit contenant les templates *paster* pour créer des installations et produits plone.

Avec Plone 3 il faut utiliser paster-2.4 pour créer les templates

Au besoin il faudrat installer ZopeSkel avec easy_install (easy_install-2.4 pour python 2.4)

Ensuite on peut avoir la liste des templates :

```
paster2.4 create --list-templates
```

Pour un produit plone on utilise *plone* :

```
paster2.4 create -t plone myproject
```

Autres squelettes obsoletes

ATMusclor définit un module Musclor, contenant deux types de contenus basiques (`MusclorFolder` et `MusclorDocument`) et un *tool* (`MusclorTool`).

URL : https://svn.pilotsystems.net/projets/ATMusclor/trunk/

5.2. Règles de nommage

Références standards pour python

PEP 8 : Style Guide for Python Code

PEP 257 : Docstring Conventions

Variables, méthodes, fonctions et classes

La première règle de nommage est d'être homogène. La deuxième règle de nommage est d'être homogène.

La charte de nommage de Python est un bon début (PEP 8 : Style Guide for Python Code). Exemple :

```
class MyClass:
    def myMethod(self, my_param, mySortFunc):
        return my_param
```

Permissions dans Zope

Une permission définie par un produit *Product* se nomme *Product: View, Product: Add blobs* (notez les majuscules, les espaces, etc.) Cela permet de la retrouver plus facilement.

Découpage du projet

Au niveau des modules et fichiers, les éléments sont rassemblés par grande aspects :

On met :

- les contenus dans *content* (donc le modèle)
- ce qui concerne l'interface dans *browser* (donc les vues)
- les profiles generic setup dans *profiles* (donc la configuration)

Ensuite le code plone récent privilégie un rassemblement fonctionnel des éléments. Ainsi si j'ai un traitement sur un évènement spécifiques à un type de contenu pour traiter des spécificités métier (et non liées à l'interface) il peut-être préférable de les mettre dans le fichier qui définit le type de contenu.

Il n'y a pas de critère de choix définitif sur ces aspects. L'important est de toujours penser à la lisibilité du fonctionnement de l'application pour un autre développeur.

5.3. Protection des méthodes dans Zope

Note : on pense à la sécurité dès la conception du code, sinon c'est le meilleur moyen pour laisser des trous béants derrière soi.

Code CMF / Zope 2

Dans toute classe, il faut :

- définir une protection sur toutes les méthodes, y compris celles qui commencent par un _ ;
- protéger au maximum les méthode :
- ne déclarer *public* que celles qui doivent vraiment l'être ;
- déclarer *private* celles qui sont a priori à usage interne ;
- déclarées *protected* les autres, en choisissant une permission appropriée : si une permission existe déjà (*View*, *Manage portal*, etc.), l'utiliser (en l'important depuis le module où elle est définie, de préférence : par exemple `from Products.CMFCore.permissions import ManagePortal`). Sinon, en créer une (cf ReglesDeNommage).

Oublier une de ces règles est le meilleur moyen de :

- se retrouver avec une méthode accessible aux anonymes, alors qu'on n'a pas vraiment envie ;
- renommer une méthode `_foo()` en `foo()` et oublier de la protéger. Dans ce cas, si l'on avait laissé `security.declareProtected("View, "_foo")'`, Zope nous aurait averti que `_foo` n'existe pas.

Code Zope 3

C'est le zcml qui va déclarer les permissions sur les vues. En dehors de ça il n'y a pas de contrôle de sécurité au niveau accès aux méthodes.

Par contre bien utiliser les conventions python pour mettre en évidence les attributs ou méthodes non publiques.