tutoriel

Internationalisation : traduire un programme Python

Il s'agit d'adapter une application (en particulier son interface basée sur glade), écrite en Python dans une langue donnée, dans une autre langue utilisateurs.

La traduction d'un programme se déroule en deux phases :

la préparation du programme : on écrit le programme en anglais (généralement), et on marque tous les textes et données spécifiques à l'anglais ou au Royaume-Uni

la traduction et l'adaptation à chaque langue ou pays.

Pour cela, nous utiliserons les outils GNU gettext, et le module python gettext.

Les outils GNU gettext, fournis avec certains systèmes comme les distributions GNU/Linux, permettent de réaliser les opérations des développeurs et des traducteurs mais sont inutiles sur la machine de l'utilisateur, où le programme n'a besoin que du module python gettext, fourni en standard avec la distribution python.

Dans tous les cas, les outils GNU gettext sont recommandés, ce tutoriel étant fait pour eux.

Pré-requis

Première étape : marquage des chaînes à traduire

Dans les différents modules python, les chaînes de caractère à traduire sont passées comme argument à une fonction, nommée par convention _ (le caractère de soulignement).

Cas d'une chaîne simple

Par exemple :

```
print "Hello world"
```

va devenir :

```
print _("Hello world")
```

Chaînes contenant des parties variables

Il ne faut pas écrire :

```
print _("Hello ") + name + _("!")
```

qui obligerait le traducteur à traduire d'abord “Hello ” (avec une espace), puis “!”, mais plutôt :

```
print _("Hello %s!") % name
```

Chaînes contenant plusieurs parties variables

Dans ce cas, nommez-les et utilisez un dictionnaire et non une liste, ce qui permettra au traducteur d'intervertir les variables.

Par exemple, ne pas écrire :

```
print _("Hello %s, today is %s") % (name, day)
```

mais plutôt :

```
print _("Hello %(name)s, today is %(day)s") % {'name': name, 'day': day}
```

Attention aux formats de nombre, de date, de monnaie, qui dépendent du pays de l'utilisateur.

Si vous utilisez **glade** pour réaliser votre interface, vous pouvez choisir les textes qui ne doivent pas être traduits (par défaut, ils sont tous traduisibles).

Autres étapes

Modification du programme python

Dans cette seconde étape, nous utilisant le module **gettext** de python pour définir la fonction **_()**.

C'est plus facile sous GNU/Linux que sous Microsoft Windows. Nous donnons un exemple pour GNU/Linux et un exemple multi-plateforme.

Dans les exemples, l'application appelée *coincoin* est lancée par *coincoin.py*, et son interface glade est contenue dans le fichier *coincoin.glade*.

Seul le fichier qui sera exécuté (*coincoin.py*) doit être modifié. S'il importe des modules avec des chaînes marquées par **_()**, celles-ci seront aussi traduites.

Les localisations sont stockées dans un dossier locale. Dans les systèmes GNU, le dossier **/usr/share/locale** est détecté automatiquement par le module **gettext**. On peut aussi le préciser (ici, le sous-dossier locale du dossier dans lequel se trouve le programme qui s'exécute).

Exemple pour les systèmes GNU/Linux :

coincoin.py

```
application = 'coincoin'
import gettext
gettext.install(application)
[...]
# Si votre programme utilise glade, précisez bien le domaine
gettext application: gui = gtk.glade.XML(fname="coincoin.glade",
domain=application)
```

Exemple multi-plateforme :

coincoin.py

```
application = 'coincoin'
import gettext
if os.name == 'nt':
    # Code pour Microsoft Windows

    # Chemin du dossier locale sous windows
    win_local_path = os.path.abspath
    (os.path.join(os.path.pardir, 'locale'))

    # Code pour une éventuelle interface glade
    gtk.glade.bindtextdomain(application, win_local_path)
    gtk.glade.textdomain(application)

    # Code pour le programme python (le module
    #local permet de déterminer la langue actuelle)
    import locale
    lang = locale.getdefaultlocale()[0][:2]
    try:
        cur_lang = gettext.translation(application,
         localedir=win_local_path, \languages=[lang])
        cur_lang.install()
    except IOError:
        # Si la langue locale n'est pas supportée,
        #on définit tout de même _()
        _ = lambda text:text
        # S'il existe des chaînes traduisibles dans
        #d'autres modules, normalement gérés par
```

```
        # cur_lang.install() ou gettext.install(),
        #vous devez aussi y définir _(). Exemple :
        sous_module._ = _
else :
    # Code pour les autres systèmes d'exploitation
    gettext.install(application)

[...]

# Si votre programme utilise glade, précisez bien le
#domaine gettext application:
gui = gtk.glade.XML(fname="coincoin.glade", domain=application)
```

Il reste à créer un modèle de traduction qui servira à créer ou mettre à jour de nouvelles traductions.

L'outil utilisé est **xgettext**. Dans un terminal, allez dans le dossier où sont stockés les modules python et les fichiers glade et lancez la commande :

- `xgettext -k_ -kN_ -o coincoin.pot *.py *.glade`

Le fichier **coincoin.pot** obtenu est le modèle des futures traductions. À chaque changement du programme, il faudra le mettre à jour en relançant cette commande.

Localisation

Cette partie doit être faite pour chaque langue.

Voici ce qu'un traducteur devra faire :

Créer une nouvelle localisation, par exemple en utilisant **msginit**. Dans un terminal et dans le dossier où se trouve **coincoin.pot**, lancer la commande :

```
msginit -i coincoin.pot -o locale/fr/LC_MESSAGES/coincoin.po
```

Ici, **locale** représente le dossier *locale* du programme et **fr** le code de la langue (les caractères avant le _ dans $LANG). Le programme msginit crée une nouvelle traduction basée sur la langue actuelle de l'utilisateur et demander quelques informations comme l'adresse mail du traducteur.

Ouvrir le fichier **coincoin.po** avec poedit et traduire les différentes chaînes de caractères.

Si le programme a été modifié depuis la création de la traduction, le traducteur devra mettre à jour son fichier depuis le modèle pot (que vous aurez pris soin de recréer) à l'aide de la commande (lancée depuis le dossier où se trouve le modèle) :

```
msgmerge -U locale/fr/LC_MESSAGES/coincoin.po coincoin.pot
```

Cette commande commente (par ajout de dièse # en début de ligne) les traductions obsolètes et

ajoute les nouvelles chaînes à traduire. Le fichier devra être relu et complété.

Compiler la traduction pour que le programme puisse l'utiliser. Cela s'effectue par la commande (dans le dossier où se trouve le fichier po) :

```
msgfmt coincoin.po -o coincoin.mo
```

Le fichier **mo** produit doit être distribué dans les paquets binaires de l'application, dans l'arborescence correspondante du dossier local de votre programme.

Conclusion

Problèmes connus

Voir aussi

- **(fr)** Tutoriel : traduire un programme en python (et glade)
- **(fr)** Internationaliser ses programmes Python, par nicolargo
- **(fr)** Traduire un programme en python
- **(fr)** article Framasoft
- **(fr)** Tutoriel vidéo (exemple en PHP)
- **(fr)** http://herverenault.fr/aide-mémoire-PHP-Perl-Python-JavaScript-Java.html
- **(fr)** Cours de Python (Paris 7)
- **(fr)** Penser en Tkinter, par Stephen Ferg
- **(fr)** Pygtk et Glade
- **(fr)** Votre première application graphique avec Python et Glade
- **(fr)** Créer des interfaces graphique avec PyGTK et Glade - Chap.1
- **(fr)** Chap. 2
- **(en)** How do I internationalize a PyGTK and libglade program? (FAQ PyGTK)
- **(en)** Documentation Python-internationalization

Basé sur « Article » par Auteur.