Interfaces graphiques pour Python avec wxPython

Alain Delgrange

"La connaissance utile, c'est celle qu'on partage..."

Sommaire

Introduction....4
Pré-requis....4
Premier programme (Bonjour tout le monde)....4
- Fichier : Exemple1.py....5
  - Ligne 01....5
  - Ligne 02....5
  - Ligne 04....5
  - Lignes 06 à 11....6
  - Lignes 13 à 18....10
  - Lignes 20 et 21....10

Gérer les évènements....12
- Fichier : Exemple2.py....12
  - Lignes 09, 10 et 11....13
  - Ligne 12....13
  - Lignes 14 à 16....14
- Fichier : Exemple3.py....15

Utilisation des boites de placement (sizers)....16
- wx.BoxSizer....16
  - Fichier : Exemple4.py....17
  - Ligne 09....18
  - Ligne 10....18
  - Ligne 13....18
  - Ligne 14....18
  - Ligne 15....19
  - Ligne 16....19
  - Ligne 17....19
  - Fichier : Exemple5.py....19
- wx.GridSizer....21
  - Fichier : Exemple7.py....21

Les Menus....22
- Fichier : Exemple8.py....22
  - Ligne 9....23
  - Lignes 10, 11 et 13....23
  - Ligne 12....24
  - Ligne 14....24
  - Ligne 15....24
  - Ligne 16....24

Lier les éléments d'un menu à des méthodes....25
- Fichier : Exemple9.py....25

Lignes 11, 12 et 14....................26
Ligne 20....................26
Ligne 21....................26
Ligne 22....................26
Ligne 23....................27
Lignes 25 à 27....................27
Lignes 29 à 36....................27
Construisons une véritable application....................27
Code de l'application :....................28
Les variables globales....................31
Lignes 6 et 7....................31
Lignes 9 à 13....................32
Construction de la fenêtre principale....................32
La barre d'outils 32
Ligne 80....................32
Lignes 82 à 107....................32
Ligne 108....................33
Ligne 109....................33
Le corps de la fenêtre....................34
La classe Visu....................34
La classe wx.ScrolledWindow....................34
La méthode Affiche()....................34
Lignes 25 à 31....................35
Lignes 32 à 37....................35
Mise en oeuvre des fonctionnalités....................35
La méthode OnOpen(self, evt)....................35
La méthode Plus(self, evt)....................39
La méthode Moins(self, evt)....................40
La méthode Retour(self, evt)....................41
La méthode OnClose(self, evt)....................41
La méthode OnExit(self, evt)....................41
La classe App()....................41
Conclusion....................42
Licence 42

Introduction

wxPython est une collection de modules Python réalisée sur la base des wxWidgets de wxWindows, un framework multi-plateformes écrit en C++.

wxPython met à disposition des développeurs un nombre impressionnant de classes permettant de réaliser des interfaces homme machine (IHM) complètement indépendantes de l' OS sur lequel ils sont mis en oeuvre.

Mais au delà, d'une simple IHM, wxPython fournit également des classes de haut niveau aussi diverses que les sockets réseau, le traitement de l'image ou encore l'interprétation HTML ou XML.

Le seul inconvénient que j'ai pu trouver dans wxPython, c'est la difficulté, notamment pour quelqu'un qui connaît mal l'anglais, d'en approcher la documentation, qui, bien que très complète, est particulièrement mal adaptée à l'apprentissage.

Le but de ce tutoriel n'est donc pas de décrire l'ensemble des classes wxPython, mais plutôt de donner des clés permettant au débutant de comprendre la philosophie des wxWidgets, et de naviguer facilement dans leur documentation.

Ce tutoriel n'est pas non plus destiné à enseigner Python, dont la connaissance est un préalable pour pouvoir aborder wxPython.

Tout au long de mes exemples, je vous renverrai sur la documentation anglaise extrêmement bien construite des wxWidgets. Vous verrez qu'à force, vous parviendrez à vous y retrouver, et en ferez peu à peu votre référence unique, bien que celle-ci soit réalisée à destination des programmeurs en C++.

Il existe bien une documentation des classes spécifique à Python, mais moins bien articulée, et qui ne sera réellement utile qu'aux programmeurs connaissant déjà bien le produit.

Pré-requis

Vous devez avoir Python installé sur votre machine (personnellement c'est la version 2.4).

Vous devez récupérer wxPython que vous trouverez sur le site http://www.wxpython.org. Faites attention de bien récupérer la version de wxPython qui correspond à celle du Python installé sur votre machine.

Pour l'installation, utilisez la procédure qui correspond à votre système d'exploitation.

Sous MS Windows l'installation est simplifiée par un programme qui automatise les procédures.

Sous linux, il existe des packages pré-compilés pour les plus courantes des distributions. Personnellement, je travaille avec la distribution Ubuntu Hoary basée sur la Debian, et le système d'installation très performant de cette distribution est complètement transparent pour l'utilisateur.

Pour le codage, on peut utiliser un éditeur de texte courant.

Personnellement j'utilise tout simplement Idle qui est fourni avec Python.

Premier programme (Bonjour tout le monde)

Je ne résisterai pas au plaisir de vous initier au toujours populaire «Bonjour tout le monde !» (« Hello world ! » en anglais) que tout enseignant d'un langage informatique se fait un devoir de servir à ses élèves...

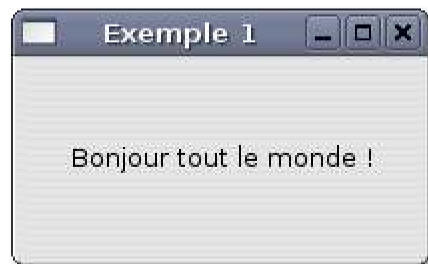
Nous allons utiliser quatre classes de wxPython pour réaliser cet exemple:

- La classe **Frame** qui fournit une fenêtre classique.
- La classe **Panel** qui fournit un conteneur polyvalent pour y déposer des contrôles.
- La classe **StaticText** qui fournit une étiquette sur laquelle on peut afficher du texte.
- La classe **App** qui est la classe de base de toute application wxPython.

Fichier : Exemple1.py

```
01 # !/usr/bin/python
02 # -*- coding: iso-8859-15 -*-
03
04 import wx
05
06 class Bonjour(wx.Frame):
07   def __init__(self, titre):
08     wx.Frame.__init__(self, None, -1, title = titre, size = (200, 100))
09     conteneur = wx.Panel(self, -1, size = self.GetClientSize())
10     etiquette = wx.StaticText(conteneur, -1, "Bonjour tout le monde !",
                                style = wx.ALIGN_CENTRE)
11     etiquette.CentreOnParent()
12
13 class MonApp(wx.App):
14   def OnInit(self):
15     fen = Bonjour("Exemple 1")
16     fen.Show(True)
17     self.SetTopWindow(fen)
18     return True
19
20 app = MonApp()
21 app.MainLoop()
```

Résultat à l'exécution :



Le visuel de cette fenêtre peut être très différent sur votre machine, selon que son OS est MS Windows, Linux (selon le thème du bureau), ou MAC OS. La capture qui précède a été réalisée sous Linux Ubuntu Hoary en environnement gnome.

Décortiquons un peu ces lignes de code :

Ligne 01

Cette ligne est indispensable sous Linux pour indiquer à l'OS où se situe l'interpréteur Python, et pour permettre un lancement du script par double-clic.

Ligne 02

Cette ligne permet de faire prendre en compte par l'interpréteur les accentués et le sigle € .

Ces deux premières lignes figurent toujours en début de mes scripts Python.

Ligne 04

Depuis la version 2.4 de wxPython, l'espace de nom du module a changé, et se nomme tout simplement wx (l'ancien espace de nom – wxPython.wx – n'a été maintenu que pour assurer la compatibilité ascendante). Il suffit donc d'importer wx pour accéder aux wxWidgets. On

pourrait également les importer sous la forme « `from wx import *` » afin d'éviter de systématiquement devoir préfixer chaque objet utilisé, mais je préfère personnellement utiliser le préfixe wx qui me permet de repérer immédiatement dans le code les références à wxPython.

Lignes 06 à 11

On construit une classe Bonjour dérivée du wxWidget wx.Frame (**ligne 06**) dont une instance sera la fenêtre qu'on veut afficher à l'écran.

Pour connaître les paramètres attendus par une wx.Frame, il suffit d'aller sur le site wxpython.org à l'adresse http://www.wxpython.org/onlinedocs.php (sous MS Windows, l'intégralité de cette documentation est installée avec wxPython sur votre disque local) et choisir ***alphabetical class reference*** puis ***wxFrame***.

Vous arrivez alors sur la documentation wxFrame où vous voyez que ce widget dérive de :

- wxTopLevelWindow
- wxWindow
- wxEvtHandler
- wxObject

Vous voyez aussi que sont listés un certain nombre de styles de fenêtre, et de processus d'événements par défaut.

Retenez dans un premier temps, que les styles de fenêtre sont des styles supplémentaires par rapport à ceux de la classe parente wxWindow, et que les processus d'événements sont également supplémentaires par rapport à la classe parente wxEvtHandler.

La rubrique intitulée ***Members*** liste les méthodes implémentées dans la classe.

Celle qui nous intéresse est le constructeur de la classe dénommé ***wxFrame::wxFrame*** tel que défini en C++.

Si vous cliquez sur ce lien, vous arrivez à la documentation de cette méthode:

wxFrame()

Default constructor.

wxFrame(wxWindow* parent, wxWindowID id, const wxString& title, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxDEFAULT_FRAME_STYLE, const wxString& name = "frame")

Il s'agit d'une méthode décrite à destination des programmeurs en C++. Il vous faut donc la transposer en méthode Python pour pouvoir l'utiliser.

Cela donne ce qui suit :

wx.Frame(parent, id, title, pos = wx.DefaultPosition, size = wx.DefaultSize, style = wx.DEFAULT_FRAME_STYLE, name = 'frame')

Signification des paramètres :

parent

Il s'agit de la fenêtre qui doit accueillir la wx.Frame (le conteneur). ***None*** pour une wx.Frame en TopLevel

id

Un entier qui représente le Handle de la fenêtre. Indiquer la valeur -1 pour une attribution automatique par l'interpréteur.

title

Une chaîne de caractère qui sera affichée dans la barre de titre de la fenêtre.

pos

La position de la fenêtre (coordonnées supérieures gauche). La constante wx.DefaultPosition est prise par défaut, et correspond au tuple (-1, -1), qui est interprété par Python en fonction de l'OS.

size

La taille de la fenêtre (largeur, hauteur). La constante wx.DefaultSize est prise par défaut, et correspond au tuple (-1, -1) qui laisse le choix à l'interpréteur d'une taille adaptée à l'OS.

style

Le style de la fenêtre, à choisir parmi les styles spécifiques à wx.Frame ou ceux de wx.Window dont wx.Frame dérive. Le style wx.DEFAULT_FRAME_STYLE pris par défaut correspond à l'assemblage des constantes de style ***wx.MINIMIZE_BOX | wx.MAXIMIZE_BOX | wx.RESIZE_BORDER | wx.SYSTEM_MENU | wx.CAPTION | wx.CLOSE_BOX | wx.CLIP_CHILDREN.*** A noter que pour indiquer un assemblage de styles, il faut séparer chacune des constantes choisies par le trait vertical (Alt Gr + 6).

name

Le nom de la fenêtre. Je n'utilise pour ma part jamais ce paramètre qui peut être éventuellement utile sous MAC OS.

C'est donc **aux lignes 07 et 08** que nous indiquerons les paramètres souhaités.

à la **ligne 07**, dans la méthode __init__() (habituelle des classes Python) de la classe Bonjour, nous indiquons les paramètres attendus pour instancier la classe. Ce peut être des paramètres attendus par la classe parente (wx.Frame), ou des paramètres nouveaux, propres à la classe dérivée.

En l'occurrence, nous n'attendons que le paramètre «titre», qui correspond au paramètre «title» de wx.Frame. Les autres paramètres obligatoires ou que nous voulons alimenter par défaut pour wx.Frame seront indiqués à la **ligne 08** dans la méthode d'initialisation __init__() de la classe parente.

Hormis le mot clé «self» que nous retrouvons systématiquement en tête des arguments de méthodes de classes en Python, et qui établit explicitement un rapport entre la méthode et son propriétaire, nous avons défini par défaut que l'instance de wx.Frame créée par l'intermédiaire de la classe Bonjour ne devait pas avoir de parent (None), que son id devait être créé automatiquement (-1), que le texte à afficher dans sa barre de titre devait provenir du paramètre «titre» attendu pour sa classe fille, et que sa taille devait être d'une largeur de 200 pixels et d'une hauteur de 100 pixels. Les autres paramètres de cette Frame étant ceux définis par défaut.

Ensuite, il nous reste à y placer le texte que nous voulons y voir affiché (Bonjour le monde !).

Pour ce faire, à la **ligne 09**, nous créons, dans un premier temps, un conteneur destiné à recevoir le texte, sur la base d'un panneau wx.Panel, à placer dans notre fenêtre.

Là encore, pour connaître les paramètres attendus par un wx.Panel, il suffit d'aller sur le site wxpython.org à l'adresse http://www.wxpython.org/onlinedocs.php à la rubrique ***wxPanel***.

On voit là que le wx.Panel dérive de :

- wxWindow
- wxEvtHandler
- wxObject

La rubrique ***Members*** donne accès au constructeur ***wxPanel::wxPanel*** qui se présente comme suit:

wxPanel()

Default constructor.

wxPanel(wxWindow* parent, wxWindowID id = -1, const wxPoint& pos = wxDefaultPosition, const wxSize& size = wxDefaultSize, long style = wxTAB_TRAVERSAL, const wxString& name = "panel")

Transposé en Python, ça donne ceci :

wx.Panel(parent, id, pos = wx.DefaultPosition, size = wx.DefaultSize, style = wx.TAB_TRAVERSAL, name = 'panel')

Signification des paramètres :

parent

Il s'agit de la fenêtre qui doit accueillir le wx.Panel (le conteneur)

id

Un entier qui représente le Handle du Panel. Indiquer la valeur -1 pour une attribution automatique par l'interpréteur.

pos

La position du Panel (coordonnées supérieures gauche). La constante wx.DefaultPosition est prise par défaut, et correspond au tuple (-1, -1), qui est interprété par Python en fonction de l'OS.

size

La taille du Panel (largeur, hauteur). La constante wx.DefaultSize est prise par défaut, et correspond au tuple (-1, -1) qui laisse le choix à l'interpréteur d'une taille adaptée à l'OS.

style

Le style du Panel, à choisir parmi les styles de la classe parente wx.Window (aucun style spécifique n'a été défini pour un wx.Panel). La constante de style wx.TAB_TRAVERSAL prise par défaut signifie que la touche TAB pourra être utilisée pour passer de contrôles en contrôles (pour les contrôles posés sur le Panel).

name

Le nom de la fenêtre. Je n'utilise pour ma part jamais ce paramètre qui peut être éventuellement utile sous MAC OS.

Nous voyons donc que nous créons un wx.Panel dont l'identificateur est ***conteneur***, le conteneur (parent) est le wx.Frame (référencé par le mot clé ***self***), l'id est -1 (attribution automatique), et la taille (size) est calculée par la méthode ***GetClientSize()*** de la wx.Frame (mot clé ***self***).

Si on se réfère à la documentation à la rubrique wxFrame, on ne trouve aucune méthode de ce nom. C'est là l'avantage de la programmation orienté objet ; une classe hérite des méthodes de ses classes parentes. La plupart des méthodes de base des wxWidgets sont héritées de la classe wxWindow. On va donc consulter la documentation à la rubrique wxWindow, et on découvre alors la méthode ***wxWindow::GetClientSize***.

wxSize GetClientSize() **const**

This gets the size of the window 'client area' in pixels. The client area is the area which may be drawn on by the programmer, excluding title bar, border, scrollbars, etc.

En français :

Cette méthode retourne la taille de la zone 'client' de la fenêtre, en pixels. La zone client est la zone sur laquelle le programmeur peut dessiner, à l'exclusion de la barre de titre, des bordures, des ascenseurs, etc.

Parameters

width (largeur)

Receives the client width in pixels.

height (hauteur)

Receives the client height in pixels.

wxPython note: In place of a single overloaded method name, wxPython implements the following methods:

GetClientSizeTuple()	Returns a 2-tuple of (width, height)
GetClientSize()	Returns a wxSize object

En détaillant sa documentation, on voit qu'il y a une particularité wxPython à cette méthode.

Elle a été implémentée de deux façons différentes:

GetClientSizeTuple() qui retourne un tuple composé de deux données, la largeur et la hauteur.

Cette méthode peut être très intéressante lorsqu'on a besoin de récupérer indépendamment l'une de l'autre les deux données. On peut alors le faire de la façon suivante :

```
largeur, hauteur = fenetre.GetClientSizeTuple()
```

GetClientSize() qui retourne les dimensions sous forme d'un objet wx.Size (il s'agit d'un objet englobant de façon monolithique les dimensions) pouvant être passé directement en paramètre size d'une fenêtre.

C'est donc cette dernière méthode que nous utilisons à la **ligne 09**, de façon à ce que notre wx.Panel prenne toute la place de la zone client de la wx.Frame.

On passe ensuite au message proprement dit qu'on veut afficher dans notre fenêtre.

A la **ligne 10**, on crée un contrôle ***wx.StaticText***, qui correspond à une simple étiquette sur laquelle on positionne le texte voulu.

On peut aller voir sa documentation en ligne, toujours au même endroit, et on peut changer de type de recherche en choisissant une recherche par catégories, puis les contrôles, et parmi ceux-ci on clique sur ***wxStaticText***.

On voit que cet objet dérive de :

- wxControl
- wxWindow
- wxEvtHandler
- wxObject

Son constructeur en C++ est le suivant:

wxStaticText()

Default constructor.

wxStaticText(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxString&** *label*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = 0*, **const wxString&** *name* = "*staticText*")

transposé en Python ça donne:

wx.StaticText(parent, id, label, pos = wx.DefaultPosition, size = wx.DefaultSize, style = 0, name = 'staticText')

Signification des paramètres :

parent

Il s'agit de la fenêtre qui doit accueillir le wx.StaticText (le conteneur). Comme nous plaçons ce contrôle sur notre wx.Panel, ce paramètre est passé sa référence, soit 'conteneur'.

id

Un entier qui représente le Handle du contrôle. Indiquer la valeur -1 pour une attribution automatique par l'interpréteur.

label

La chaîne de caractères que nous voulons voir affichée ('Bonjour le monde !').

pos

La position du contrôle (coordonnées supérieures gauche). La constante wx.DefaultPosition est prise par défaut, et correspond au tuple (-1, -1), qui est interprété par Python en fonction de l'OS.

size

La taille du contrôle (largeur, hauteur). La constante wx.DefaultSize est prise par défaut, et correspond au tuple (-1, -1) qui laisse le choix à l'interpréteur d'une taille adaptée à l'OS.

style

Le style de la fenêtre, à choisir parmi les styles spécifiques à wx.StaticText ou ceux de wx.Window dont wx.StaticText dérive. Les styles spécifiques du contrôle sont des styles de présentation du texte dans le contrôle : ***wx.ALIGN_RIGHT*** aligne le texte sur le bord droit du contrôle, ***wx.ALIGN_LEFT*** aligne le texte sur le bord gauche du contrôle, et ***wx.ALIGN_CENTRE*** place le texte au centre du contrôle (c'est ce dernier style que nous utilisons)...

Il existe un style supplémentaire, ***wx.ST_NO_AUTORESIZE***, qui empêche le contrôle de changer de taille quand le texte est modifié à l'aide de la méthode **SetLabel()**.

name

Je ne reviens pas sur ce paramètre.

Une fois notre contrôle texte créé, il nous reste à le centrer dans notre fenêtre. On fait ça à la **ligne 11** à l'aide de la méthode CentreOnParent(), dérivée de la classe wxWindow et qui permet de centrer une fenêtre ou un contrôle dans son conteneur. Cette méthode prend un paramètre initialisé par défaut avec la constante ***wx.BOTH*** qui signifie un centrage à la fois vertical et horizontal. les deux autres valeurs acceptées sont ***wx.VERTICAL*** et ***wx.HORIZONTAL***, qui n'ont pas besoin de précisions complémentaires.

Lignes 13 à 18

C'est là la particularité de wxPython, celle qui nous oblige à pratiquement toujours développer en pur objet.

Pour pouvoir lancer un interface graphique wxPython, on doit obligatoirement passer par un objet chargé de créer l'instance de la classe principale de l'interface (le Top Level).

Cet objet doit lui-même être construit et dériver de la classe ***wx.App*** ; c'est ce qu'on fait à la **ligne 13**.

La classe wx.App est initialisée différemment des classes Python classiques. Ce n'est pas la méthode __init__() qui doit être implémentée, mais une méthode ***OnInit()*** sans paramètre (sauf le mot clé self qui lui permet de s'auto-référencer) ; c'est ce qui est fait à la **ligne 14**.

La séquence d'initialisation de la méthode ***OnInit()*** est alors toujours la même :

- Création d'une instance de la fenêtre principale (**ligne 15**)
- Affichage de la fenêtre par la méthode ***Show()*** dérivée de la classe ***wx.Window*** (**ligne 16**)
- Désignation de la fenêtre en tant que principale par la méthode ***SetTopWindow()*** spécifique à la classe ***wx.App*** (**ligne 17**)
- Retour de la valeur ***True*** marquant la fin de l'initialisation (**ligne 18**).

Lignes 20 et 21

C'est là que démarre réellement notre application.

A la ***ligne 20,*** on crée une instance de l'application, qui elle-même crée une instance de notre fenêtre principale (classe Bonjour).

A la **ligne 21** on démarre la boucle de gestion des évènements, qui permet au programme d'inter-agir avec son environnement.

Résumé

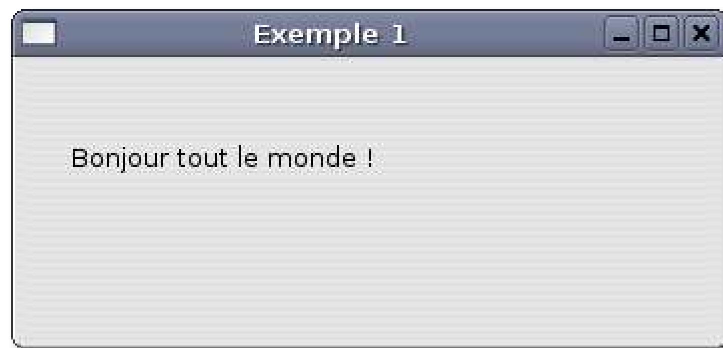
On a vu dans ce chapitre:

- Comment créer une fenêtre en wxPython
- Comment positionner un contrôle dans une fenêtre
- Comment consulter la documentation en ligne
- Comment créer une application basée sur wxPython

C'est loin d'être suffisant, mais c'est un bon début...

Gérer les évènements

Notre premier programme est magnifique d'équilibre, mais que ce passe-t-il lorsqu'on dimensionne notre fenêtre à la souris ?



Pas terrible n'est-ce pas ? Notre texte n'est plus centré...

Pourquoi ? Tout simplement parce que nous n'avons pas prévu ce qui devait se passer quand l'utilisateur décide de modifier les dimensions de la fenêtre.

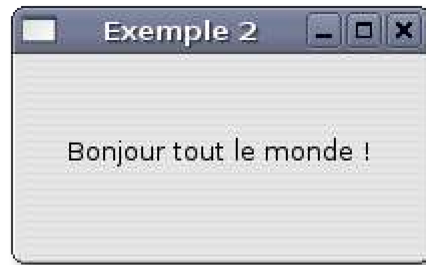
Pour pouvoir réagir au dimensionnement de notre fenêtre, il faut que notre programme soit prévenu de l'évènement, sache de quel type d'évènement il s'agit, et puisse effectuer le traitement adéquat.

Fichier : Exemple2.py

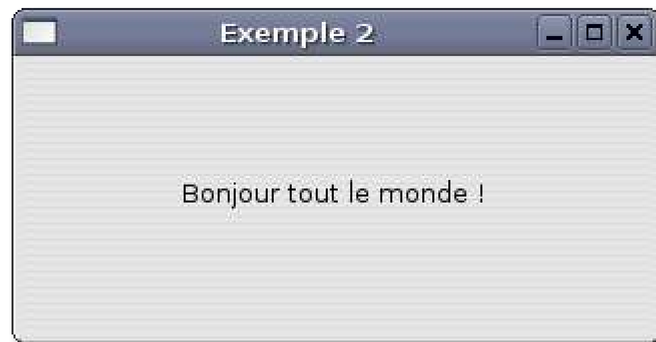
```
01 #!/usr/bin/python
02 #-*- coding: iso-8859-15 -*-
03
04 import wx
05
06 class Bonjour(wx.Frame):
07   def __init__(self, titre):
08     wx.Frame.__init__(self, parent = None, id = -1, title = titre, size = (200, 100))
09     self.conteneur = wx.Panel(self, -1, size = self.GetClientSize())
10     self.etiquette = wx.StaticText(self.conteneur, -1, "Bonjour tout le monde !",
                                      style = wx.ALIGN_CENTRE)
11     self.etiquette.CentreOnParent()
12     wx.EVT_SIZE(self, self.OnSize)
13
14   def OnSize(self, evt):
15     self.conteneur.SetSize(self.GetClientSize())
16     self.etiquette.CentreOnParent()
17
18 class MonApp(wx.App):
19   def OnInit(self):
20     fen = Bonjour("Exemple 2")
21     fen.Show(True)
22     self.SetTopWindow(fen)
23     return True
24
25 app = MonApp()
26 app.MainLoop()
```

Notre programme modifié en conséquence est listé ci-dessus (les modifications apparaissent en rouge).

La capture d'écran qui suit ne révèle, à part le titre, aucune différence par rapport au résultat du programme «Exemple1.py».



Par contre si on dimensionne la fenêtre à l'aide de la souris,



On voit que le texte reste maintenant centré dans la fenêtre.

Examinons en détail les modifications qui ont permis un tel résultat...

Lignes 09, 10 et 11

Il s'agissait tout simplement de rendre publiques les références de notre wx.Panel et de notre wx.StaticText de façon à ce qu'elles soient accessibles depuis une autre méthode que la méthode __init__().

On le fait en les transformant en membres de la classe bonjour en les préfixant du mot clé self.

Attention, à partir de là, il faut toujours les préfixer de self lorsqu'on veut les utiliser, sinon python considérera que conteneur est une référence différente de self.conteneur.

Ligne 12

C'est cette ligne qui nous permet d'intercepter l'évènement attendu, et de référencer la méthode qui en effectuera le traitement.

wx.EVT_SIZE() n'est pas une méthode, mais une macro-commande liée à la classe ***wxSizeEvent***, elle-même dérivée de la classe ***wxEvent***. Cette macro renvoie un objet ***wxEvent*** de classe ***wxSizeEvent*** dès que l'évènement se produit.

Si vous accédez à la documentation en ligne par le classement par catégories, vous trouvez, à la catégorie **Events**, une liste des classes dérivée de ***wxEvent***, dont notre classe ***wxSizeEvent***.

En cliquant sur le lien, vous accédez au détail de notre classe :

Comme pour les autres descriptions de classes, on y voit de quelles classes elle dérive :

- wxEvent
- wxObject

Mais on a une rubrique supplémentaire appelée **Event table macros**.

On trouve dans cette rubrique l'ensemble des macros disponibles pour la classe d'évènement considérée. En l'occurrence, pour la classe **wxSizeEvent**, la seule macro disponible est ***EVT_SIZE(funct)*** qu'on peut transposer en wxPython en **wx.EVT_SIZE(objet, méthode)**.

Les paramètres attendus sont les suivants :

- **Objet** : Les évènements sont propagés depuis le contrôle jusqu'à sa classe parente. C'est donc le mot clé self qui sera toujours indiqué, puisque la classe dans laquelle est déclarée la macro est forcément parente du contrôle concerné.
- **méthode** : C'est la méthode de la classe englobant la macro, chargée de traiter l'évènement qui lui est automatiquement renvoyé par la macro dès que celui-ci se produit (dans notre exemple, c'est la méthode ***OnSize()***, préfixée ***self***, puisque c'est obligatoirement une méthode de classe qui doit être appelée).

Vous pouvez naviguer dans la documentation en ligne, parmi les classes de gestion d'évènements, il en existe pour tout ce qui peut interagir avec votre application, et elles possèdent toutes des macros vous permettant de les intercepter.

La plupart des macros qui interceptent les évènements liés à des actions de l'utilisateur (clic sur des boutons, sur des menus, appui sur des touches, etc. sont fournies par la classe wxCommandEvent. Les autres autres classes fournissent des macros relatives à des évènements plus spécifiques aux contrôles qu'ils concernent.

Le nombre de paramètres attendus par les macros dépend de la propagation de l'évènement, et de la spécificité de l'évènement attendu. Dans le cas de notre macro wx.SIZE_EVENT(), le seul paramètre attendu, hormis le mot clé self, est le nom de la méthode de classe chargée de le traiter. Il est inutile de lui passer l'Id du contrôle qui génère l'évènement puisque c'est la feuille elle-même qui est en cause. Par contre, un événement originaire d'un bouton ou du clic sur un menu, sera récupéré sous la forme wx.EVT_XXX(self, id, méthode), ou id est l'id du contrôle ayant généré l'évènement.

Lignes 14 à 16

C'est la méthode de gestion de l'évènement renvoyé par la macro. Cette méthode doit avoir le même nom que la méthode passée en paramètre de la dite macro.

A la **ligne 14**, on définit la méthode avec attente d'un paramètre '***evt***' (le mot clé ***self***, n'est là que pour indiquer qu'il s'agit d'une méthode de la classe Bonjour).

Ce paramètre, fourni par la macro, est tout simplement l'objet de classe ***wxSizeEvent*** généré par notre dimensionnement de fenêtre.

Que faire de cet objet ?

Si on retourne à la documentation en ligne à la rubrique ***wxSizeEvent***, on constate que cette classe possède, outre une macro-commande, des méthodes membres.

- Un constructeur dont vous n'avez pas à vous préoccuper(sauf si vous créez votre propre classe dérivée de wxSizeEvent, mais ça c'est une autre histoire), puisque l'objet est construit automatiquement.
- une méthode GetSize() qui renvoie la taille de la fenêtre après dimensionnement.

Dans notre exemple, nous voulons simplement adapter la taille de notre wx.Panel (self.conteneur) à notre fenêtre, et à maintenir notre wx.StaticText (self.etiquette) au centre.

Nos méthodes GetClientSize() et CentreOnParent() utilisées aux **lignes 15 et 16** sont adaptées à ces tâches, c'est pourquoi nous n'avons pas utilisé la méthode fournie par l'évènement, mais il y a un tas d'informations qu'on peut récupérer pour traiter un événement.

Outre les méthodes propres au type d'évènement attendu, il suffit d'aller voir la documentation en ligne de la classe wxEvent dont dérivent toutes les classes d'évènements pour s'en convaincre.

Ces méthodes sont particulièrement simples à utiliser.

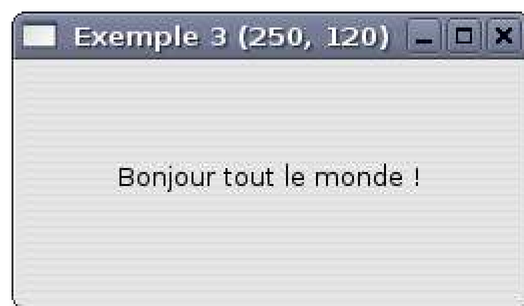
Voyons notre exemple auquel nous ajoutons une ligne (en rouge) dans la méthode OnSize() (à noter que j'ai augmenté les dimensions de la fenêtre à la ligne 08 de façon à ne pas tronquer

le titre):

Fichier : Exemple3.py

```
01 #!/usr/bin/python
02 #-*- coding: iso-8859-15 -*-
03
04 import wx
05
06 class Bonjour(wx.Frame):
07   def __init__(self, titre):
08     wx.Frame.__init__(self, parent = None, id = -1, title = titre, size = (250, 120))
09     self.conteneur = wx.Panel(self, -1, size = self.GetClientSize())
10     self.etiquette = wx.StaticText(self.conteneur, -1, "Bonjour tout le monde !",
                                       style = wx.ALIGN_CENTRE)
11     self.etiquette.CentreOnParent()
12     wx.EVT_SIZE(self, self.OnSize)
13
14   def OnSize(self, evt):
15     self.SetTitle("Exemple 3 %s" % evt.GetSize())
16     self.conteneur.SetSize(self.GetClientSize())
17     self.etiquette.CentreOnParent()
18
19 class MonApp(wx.App):
20   def OnInit(self):
21     fen = Bonjour("Exemple 3")
22     fen.Show(True)
23     self.SetTopWindow(fen)
24     return True
25
26 app = MonApp()
27 app.MainLoop()
```

Quand on lance le programme, ça donne ceci :



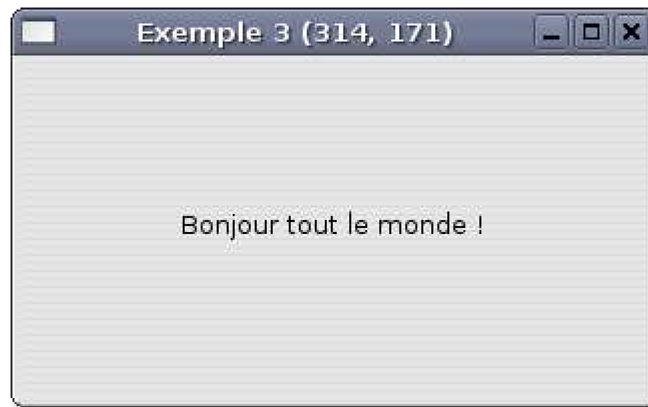
On voit que la barre de titre est modifiée et affiche la taille de la fenêtre sous la forme d'un objet ***wxSize*** de (250, 120).

Pourquoi à l'affichage, alors qu'il n'y a eu aucune tentative de dimensionnement de la fenêtre par l'utilisateur ?

Tout simplement parce que le constructeur de la classe ***wxFrame*** dimensionne la fenêtre dès la création de l'instance, et que dès lors, l'évènement se produisant, la méthode ***OnSize()*** est appelée...

La nouvelle **ligne 15** modifie le titre de la fenêtre en y ajoutant ses dimensions en pixels.

On peut vérifier que ça marche aussi en dimensionnement par l'utilisateur :



Magique, non ?

On est maintenant capable de gérer les actions de l'utilisateur sur notre programme.

Utilisation des boites de placement (sizers)

Comme la bibliothèque AWT de Java, ou Tkinter l'IHM intégré à Python, wxPython intègre des mécanismes de placement automatique des contrôles destinés à permettre de rendre l'application graphiquement indépendante de l'OS et de la résolution de l'écran sur lequel elle s'affiche.

L'autre avantage de ces mécanismes ou sizers c'est de gérer automatiquement les dimensionnements des fenêtres, sans obliger le programmeur à calculer les dimensions et positions des contrôles dans celles-ci.

La classe principale dont dérivent les sizers de wxPython est ***wxSizer***.

Deux types de Sizers sont disponibles :

wx.BoxSizer

Ce sizer permet d'empiler les éléments à placer sur une colonne, ou de les positionner les uns à la suite des autres sur une ligne(sa variante, ***wx.StaticBoxSizer***, dispose en outre d'un cadre avec étiquette).

Si on consulte la documentation en ligne à la rubrique ***wx.BoxSizer***, on voit que son constructeur (wxBoxSizer::wxBoxSizer en C++) attend un paramètre qui est le paramètre d'orientation (en colonne ou en ligne). On utilise pour l'alimenter, les constantes wx.VERTICAL et wx.HORIZONTAL.

Donc, si on désire utiliser un placement en empilage, on crée son sizer de la façon suivante :

```
monSizer = wx.BoxSizer(wx.VERTICAL)
```

Mais une fois notre sizer créé, comment l'utiliser ?

Là encore, c'est dans la documentation en ligne qu'on va trouver les méthodes de classe à mettre en oeuvre.

On a vu que ***wx.BoxSizer***, comme toutes les classes de sizer dérive de la classe de base ***wx.Sizer***. Cette classe possède une méthode ***.Add()*** qui permet d'ajouter les éléments à placer dans le sizer.

Cette méthode est déclinée selon trois prototypes :

Add(*window*, *proportion = 0*, *flag = 0*, *border = 0*, *userData = NULL*)

qui permet d'ajouter la fenêtre ou le contrôle désigné par le paramètre **window**.

Add(*sizer*, *proportion = 0*, *flag = 0*, *border = 0*, *userData = NULL*)

qui permet d'ajouter un autre sizer désigné par le paramètre **sizer**

Add(*width*, *height*, *proportion = 0*, *flag = 0*, *border = 0*, *userData = NULL*)

qui permet d'ajouter un espace vide de largeur **width** et de hauteur **height**.

Les paramètres communs de ces trois prototypes ont les significations suivantes :

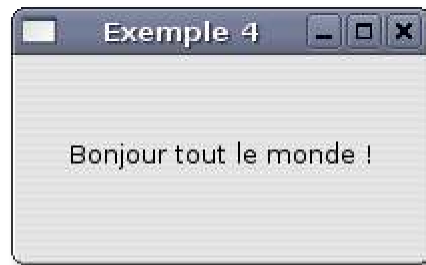
- **proportion** : ce paramètre initialisé par défaut à 0, n'est utilisé que dans le cadre du wx.BoxSizer pour indiquer que l'élément ajouté avec ce paramètre est étirable dans le même sens que l'orientation du sizer (wx.VERTICAL ou wx.HORIZONTAL) en cas de dimensionnement du sizer par action de l'utilisateur sur la fenêtre auquel il est attaché.
- **flag** : ce paramètre peut être une combinaison de constantes réalisée avec l'opérateur or (|). Deux types de constantes peuvent être combinés. Les premières se rapportent au positionnement de la bordure (dont l'épaisseur est déterminé par le paramètre border) autour de l'élément dans le sizer, (wx.ALL, wx.BOTTOM etc.). Les secondes se rapportent au comportement de l'élément en cas de dimensionnement du sizer (alignement : wx.ALIGN_XXX, élargissement avec ou sans maintien de ratio : WX.EXPAND, WX.SHAPED).
- **border** : largeur de la bordure qu'on veut maintenir autour de l'élément ajouté (nécessite un paramètre **flag** de placement).
- **userData** : Permet à un objet supplémentaire d'être attaché au sizer, pour une utilisation dans les classes dérivées (non utilisé à ma connaissance en wxPython).

Voyons maintenant comment on peut réaliser notre fenêtre «Bonjour le monde !» à l'aide de wx.BoxSizer ...

Fichier : Exemple4.py

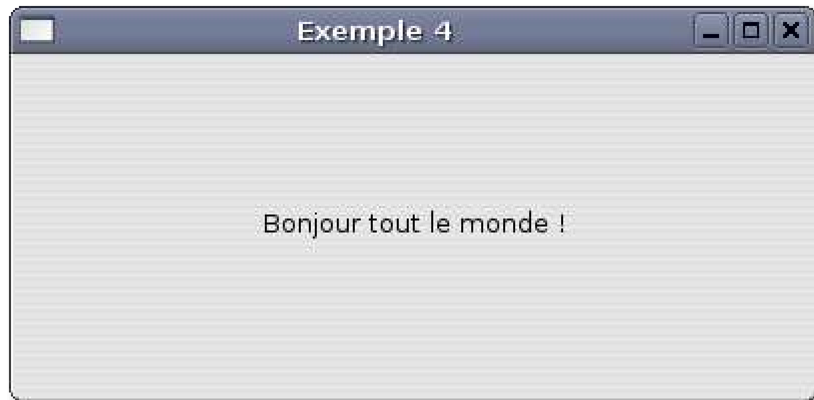
```
01 #!/usr/bin/python
02 # -*- coding: iso-8859-15 -*-
03
04 import wx
05
06 class Bonjour(wx.Frame):
07   def __init__(self, titre):
08     wx.Frame.__init__(self, None, -1, title = titre)
09     frameSizer = wx.BoxSizer(wx.VERTICAL)
10     panelSizer = wx.BoxSizer(wx.HORIZONTAL)
11     conteneur = wx.Panel(self, -1)
12     etiquette = wx.StaticText(conteneur, -1,
                          "Bonjour tout le monde !", style = wx.ALIGN_CENTRE)
13     panelSizer.Add(etiquette, 1, wx.ALIGN_CENTRE)
14     conteneur.SetSizer(panelSizer)
15     frameSizer.Add(conteneur, 1, wx.EXPAND)
16     self.SetSizer(frameSizer)
17     frameSizer.SetSizeHints(self)
18     self.SetSize(200, 100)
19
20 class MonApp(wx.App):
21   def OnInit(self):
22     fen = Bonjour("Exemple 4")
23     fen.Show(True)
24     self.SetTopWindow(fen)
25     return True
26
27 app = MonApp()
28 app.MainLoop()
```

Voici ce que cela donne à l'exécution :



Effectivement, on ne décèle aucune différence avec la fenêtre affichée à l'exemple 1.

Maintenant, voyons ce qui se passe quand la fenêtre est dimensionnée :



Bien qu'on n'ait pas intercepté l'évènement wx.SizeEvent pour dimensionner les contrôles, le texte se centre automatiquement.

Examinons les lignes de code qui en sont responsables ...

Ligne 09

On crée un wx.BoxSizer à orientation verticale destiné à positionner le wx.Panel (conteneur) sur la wx.Frame (Bonjour).

Ligne 10

On crée un autre wx.BoxSizer à orientation horizontale destiné à positionner le wx.StaticText (etiquette) sur le wx.Panel (conteneur).

Ligne 13

On ajoute le wx.StaticText au sizer du wx.Panel. On met le paramètre de proportion à la valeur 1 de façon à ce que le contrôle s'étale sur toute la largeur du sizer (qui est d'orientation horizontale) en cas de dimensionnement. Comme le style du wx.StaticText est centré, ça doit logiquement centrer horizontalement le texte dans tous les cas de figure.

On met le paramètre flag à wx.ALIGN_CENTRE qui aura pour effet de centrer le contrôle verticalement par rapport au sizer.

Ligne 14

On lie le sizer contenant le wx.StaticText au wx.Panel (conteneur), à l'aide de la méthode de la classe wxWindow SetSizer(sizer) qui attend une référence à un sizer en guise de paramètre.

Ligne 15

On ajoute le wx.Panel (conteneur) au sizer de la wx.Frame (Bonjour) . On met le paramètre de proportion à 1 de façon à ce que conteneur s'étale sur toute la hauteur du sizer (qui est d'orientation verticale) en cas de dimesionnement.

Ligne 16

On lie le sizer contenant le wx.Panel à la wx.Frame, à l'aide de la méthode de la classe wxWindow SetSizer(sizer) qui attend une référence à un sizer en guise de paramètre.

Ligne 17

Cette ligne permet de créer une contrainte au dimensionnement de la fenêtre, liée à la taille minimale d'un de ses composant.

A l'aide de cette méthode, vous décidez implicitement qu'un dimensionnement de la fenêtre ne pourra jamais avoir pour effet de donner à cette fenêtre une taille inférieure à la taille minimale requise pour l'affichage du texte (si vous exécutez le programme, vous verrez que vous ne pouvez jamais lui donner une taille ne lui permettant pas d'afficher le message en entier).

La méthode utilisée ***SetSizeHints()*** est une méthode de la classe ***wx.Sizer*** qui attend en paramètre la référence de la fenêtre parente de l'élément pour lequel une taille minimale est fixée.

Vous avez du remarquer que l'initialisation de la classe wx.Frame à la ligne 8 ne prévoit plus la fixation de la taille de la fenêtre, ce dimensionnement étant confié à la méthode SetSize de la **ligne 18**. En effet, lorsqu'un sizer est défini, la fenêtre sera affichée dans sa dimension la plus petite, quelque soit celle indiquée dans sa phase d'initialisation de wx.Frame. C'est pourquoi la dernière méthode appelée dans la méthode __init__ retaille la fenêtre aux dimensions souhaitées.

Pour être complet, on va modifier ce programme pour lui faire afficher en barre de titre la taille de la fenêtre comme on l'avait fait à l'exemple 3

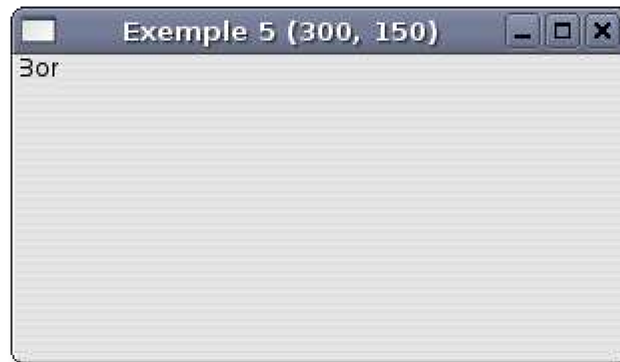
Fichier : Exemple5.py

```
01 #!/usr/bin/python
02 # -*- coding: iso-8859-1 -*-
03
04 import wx
05
06 class Bonjour(wx.Frame):
07   def __init__(self, titre):
08     wx.Frame.__init__(self, None, -1, title = titre)
09     frameSizer = wx.BoxSizer(wx.VERTICAL)
10     panelSizer = wx.BoxSizer(wx.HORIZONTAL)
11     conteneur = wx.Panel(self, -1)
12     etiquette = wx.StaticText(conteneur, -1,
                                 "Bonjour tout le monde !", style = wx.ALIGN_CENTRE)
13     panelSizer.Add(etiquette, 1, wx.ALIGN_CENTRE)
14     conteneur.SetSizer(panelSizer)
15     frameSizer.Add(conteneur, 1, wx.EXPAND)
16     self.SetSizer(frameSizer)
17     frameSizer.SetSizeHints(self)
18     self.SetSize((300, 150))
19     wx.EVT_SIZE(self, self.OnSize)
20
21   def OnSize(self, evt):
22     self.SetTitle("Exemple 5 %s" % evt.GetSize())
23
24 class MonApp(wx.App):
25   def OnInit(self):
26     fen = Bonjour("Exemple 5")
27     fen.Show(True)
```

```
28      self.SetTopWindow(fen)
29      return True
30
31 app = MonApp()
32 app.MainLoop()
```

Comme pour l'exemple 3, on intercepte l'évènement wx.SizeEvent à l'aide de la macro wx.EVT_SIZE(), et on définit la méthode OnSize() pour traiter cet événement, en modifiant la barre de titre.

Et à l'exécution ça donne ...



Catastrophe... On a bien la taille dans la barre de titre, mais notre texte n'est plus centré du tout, et il est même tronqué...

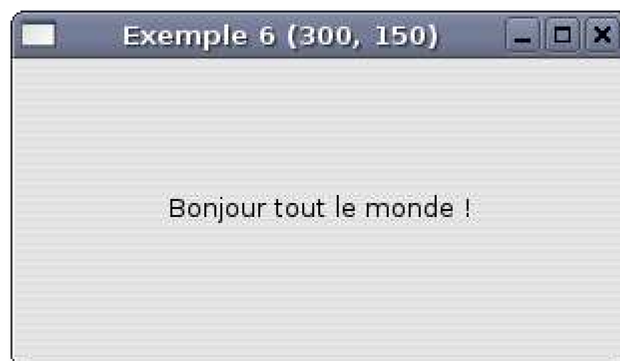
Pas d'affolement, il s'agit d'un effet tout à fait normal de l'interception de l'évènement wxSizeEvent qui est utilisé en automatique par les sizers. Le fait qu'on l'ait intercepté, l'évènement a échappé au sizer, qui n'a pas pu recalculer les positionnements.

Ce problème est très facile à résoudre. Il suffit, une fois l'évènement intercepté et utilisé, de le libérer pour d'autres gestionnaires d'évènements.

On libère un évènement en lui appliquant la méthode de la classe wxEvent Skip(), qu'on exécute en dernière ligne de la méthode de traitement de l'évènement, ce qui donne pour notre méthode OnSize() :

```
def OnSize(self, evt):
  self.SetTitle("Exemple 6 %s" % evt.GetSize())
  evt.Skip()
```

En lançant notre programme rectifié, on obtient :



Et tout est rentré dans l'ordre.

wx.GridSizer

Ce sizer permet de disposer les contrôles sur un conteneur dans une grille dont les cellules ont toutes les mêmes dimensions et dont la hauteur est égale à la hauteur de l'élément le plus haut de tous ceux qui y sont placés, et la largeur est égale à la largeur de l'élément le plus large.

Sa variante ***wx.FlexGridSizer*** est une grille dont toutes les cellules de la même ligne ont la même hauteur, et toutes les cellules d'une même colonne ont la même largeur, mais où toute les lignes n'ont pas forcément la même hauteur et toutes le colonnes n'ont pas forcément la même largeur.

Quant à son autre variante, le wx.GridBagSizer c'est un wx.FlexGridSizer qui possède sa propre méthode Add(), laquelle possède deux paramètre supplémentaire.

Le premier représente la position à laquelle on veut placer l'élément dans la grille, sous la forme d'un tuple à deux éléments (ligne, colonne).

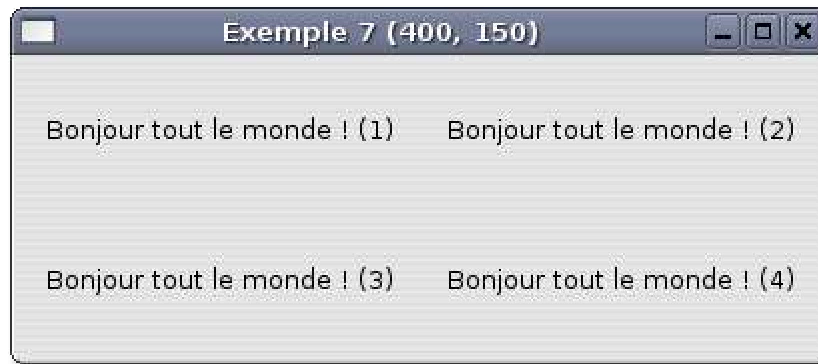
Le second permet d'indiquer la place que prendra l'élément dans la grille et est passé sous la forme d'un tuple à deux éléments (nbre de lignes, nbre de colonnes).

Voyons un exemple simple sur la base de notre bonjour le monde...

Fichier : Exemple7.py

```
#!/usr/bin/python
# -*- coding: iso-8859-1 -*-

import wx

class Bonjour(wx.Frame):
	def __init__(self, titre):
		wx.Frame.__init__(self, None, -1, title = titre)
		frameSizer = wx.BoxSizer(wx.VERTICAL)
		panelSizer = wx.GridSizer(2, 2, 0, 0)
		conteneur = wx.Panel(self, -1)
		for x in range(4):
			y = x + 1
			etiquette = wx.StaticText(conteneur, -1,
			                      "Bonjour le monde ! (%s)" % y ,
			                      style = wx.ALIGN_CENTRE)
			panelSizer.Add(etiquette, 0, wx.ALIGN_CENTRE)
		conteneur.SetSizer(panelSizer)
		frameSizer.Add(conteneur, 1, wx.EXPAND)
		self.SetSizer(frameSizer)
		frameSizer.SetSizeHints(self)
		self.SetSize((400, 150))
		wx.EVT_SIZE(self, self.OnSize)

	def OnSize(self, evt):
		self.SetTitle("Exemple 7 %s" % evt.GetSize())
		evt.Skip()

class MonApp(wx.App):
	def OnInit(self):
		fen = Bonjour("Exemple 7")
		fen.Show(True)
		self.SetTopWindow(fen)
		return True

app = MonApp()
app.MainLoop()
```

Ce qui nous donne ceci à l'exécution :



A la place du wx.BoxSizer lié au wx.Panel, on crée cette fois un wx.GridSizer de deux lignes et deux colonnes, sans espace entre cellules.

On lance ensuite une boucle qui crée quatre wx.StaticText de style centré, et les place au fur et à mesure dans le wx.GridSizer.

On voit à cette occasion, grâce à la numérotation des contrôles effectuée au fur et à mesure de leur création, que le placement se déroule ligne par ligne, et que le passage à la ligne s'effectue lorsque toutes les colonnes de la ligne ont été servies.

En utilisant un wx.GridBagSizer, on aurait pu remplir la grille dans n'importe quel ordre de cellules.

Les Menus

wxPython fournit des classes puissantes pour mettre en oeuvre des menus déroulant de façon très simple.

Ces classes, interdépendantes sont au nombre de trois :

- **wx.Menu** : Cette classe représente une liste d'éléments qui, rattachée à une barre de menus, constitue un menu déroulant, et qui utilisée seule peut constituer un menu popup.
- **wx.MenuBar** : Cette classe représente une barre de menu située au sommet d'une fenêtre. On y rattache des menus de classe ***wx.Menu***.
- **wx.MenuItem** : Cette classe représente les éléments d'un menu de classe ***wx.Menu***. En règle générale, on laisse les méthodes de la classe ***wx.Menu*** construire les ***wx.MenuItem*** pour nous.

Là encore, la documentation en ligne permet de trouver de nombreuses méthodes utiles dans la gestion des menus...

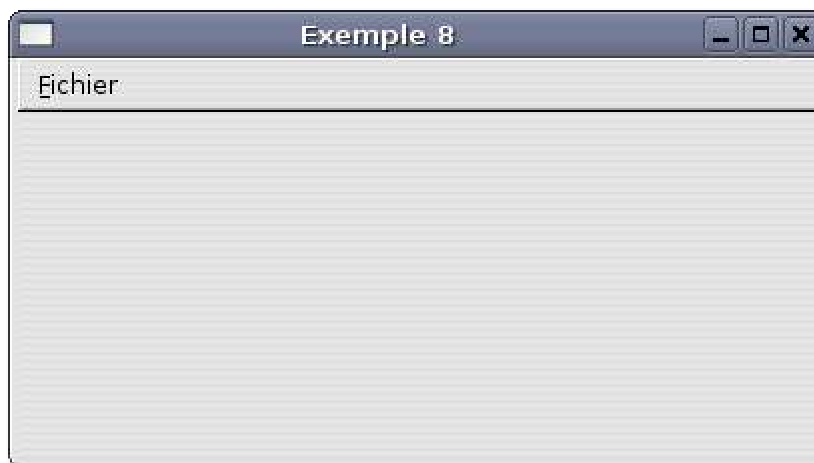
Construisons un exemple avec un menu simple.

Fichier : Exemple8.py

```
01 #!/usr/bin/python
02 # -*- coding: iso-8859-1 -*-
03
04 import wx
05
06 class LesMenus(wx.Frame):
07   def __init__(self, titre):
08     wx.Frame.__init__(self, None, -1, title = titre, size = (400, 200))
09     menuFichier = wx.Menu()
10     menuFichier.Append(wx.ID_OPEN, "&Ouvrir\tCTRL+o")
11     menuFichier.Append(wx.ID_CLOSE, "&Fermer\tCTRL+f")
12     menuFichier.AppendSeparator()
13     menuFichier.Append(wx.ID_EXIT, "&Quitter\tCTRL+q")
```

```
14     menuBarre = wx.MenuBar()
15     menuBarre.Append(menuFichier, "&Fichier")
16     self.SetMenuBar(menuBarre)
17
18 class MonApp(wx.App):
19   def OnInit(self):
20     fen = LesMenus("Exemple 8")
21     fen.Show(True)
22     self.SetTopWindow(fen)
23     return True
24
25 app = MonApp()
26 app.MainLoop()
```

qui donne à l'exécution :



Si vous cliquez sur Fichier, le menu se déroule et fait apparaître trois options «Ouvrir», «Fermer» et «Quitter», les deux premières étant séparées de la dernière par un trait horizontal.

Examinons les lignes de code ayant permis ce résultat.

Ligne 9

On crée un objet wx.Menu destiné à rassembler les éléments constitutifs du menu. Si on va voir la documentation en ligne à la rubrique wxMenu, on voit que son constructeur est le suivant :

wxMenu(**const wxString&** *title* = "", **long** *style = 0*)

soit transposé en wxPython :

wx.Menu(title = "", *style = 0)*

Dans notre code, nous avons créé l'objet sans paramètre, car le paramètre ***title*** a pour objet de mettre en première position du menu, un élément correspondant à la chaîne de caractères passée, ce qui ne présente pas d'intérêt, puisqu'on verra que c'est lors de l'ajout du menu dans la barre de menu, qu'on donnera le titre qui doit y apparaître.

Quant au paramètre ***style***, il est présent pour permettre de créer un menu détachable (avec le style wx.MENU_TEAROFF), qui n'est possible qu'avec MOTIF.

Lignes 10, 11 et 13

On ajoute les éléments du menu à l'aide de la méthode Append() de wx.Menu.

Cette méthode, prototypée de trois manières différentes en C++, est implémentée en wxPython sous la forme de trois méthodes séparées :

Append(id, string, helpStr="", checkable=false**)**

C'est la méthode classique, celle que nous avons utilisée dans l'exemple.

Explication des paramètres.

- **id** : Contrairement à ce qu'on a vu jusqu'ici, l'id d'un élément de menu ne doit surtout pas être laissé au choix de l'application à l'aide de la valeur -1, mais déterminé explicitement, et de préférence à l'aide d'une constante. Dans le cas contraire, il deviendrait impossible d'intercepter les évènements de menus (que nous verrons plus loin dans ce chapitre).

  Un nombre non négligeable de constantes ont été définies avec des noms explicites. Vous en trouverez la liste en consultant la rubrique ***«Event handling overview»*** de la documentation en ligne, au chapitre ***«Window identifiers»***.

- **string** : C'est le texte qui apparaîtra dans le menu. le caractère **&** souligne le caractère qui le suit immédiatement et permet d'utiliser la touche ALT suivi du dit caractère pour accéder directement à l'élément de menu.

  On peut également définir des raccourcis clavier liés au éléments de menu. Ces raccourcis sont composés d'une combinaison des touches CTRL, ALT ou SHIFT avec un caractère quelconque, ou une touche de fonction (F1 à F12).

  Pour définir ce type de raccourci, il suffit de compléter le paramètre string de la représentation d'une tabulation (\t) du code de la touche (CTRL, ALT ou SHIFT) , du signe + et du caractère ou du code la touche de fonction.

- **HelpStr** : Ce paramètre facultatif permet d'associer une phrase d'aide à l'élément de menu. Cette phrase d'aide apparaît dans la barre d'état de la fenêtre (quand il en existe une), dès que le curseur de la souris passe sur l'élément.

- **checkable** : Ce paramètre de type boolean est initialisé par défaut à False. Si on lui passe la valeur True, l'élément de menu devient un élément à cocher.

AppendMenu(id, string, wx.Menu, helpStr=""**)**

Cette méthode permet de lier un objet wx.Menu à l'élément de menu, de façon à créer un sous-menu accessible depuis cet élément.

AppendItem(wx.MenuItem**)**

Cette méthode permet d'ajouter un élément de menu défini séparément à l'aide de la classe wx.MenuItem.

Ligne 12

On ajoute une ligne de séparation au menu à l'aide de la méthode sans paramètre ***AppendSeparator***()

Ligne 14

On crée un objet wx.MenuBar destiné à recevoir le wx.Menu créé précedemment.

Le seul paramètre du constructeur, est le style qui est alimenté par défaut à zéro. La seule constante de style acceptée est wx.MB_DOCKABLE qui permet d'en faire une barre de menu détachable, mais elle n'a d'effet que sous GTK+.

Ligne 15

On ajoute notre menu à la barre de menu à l'aide de la méthode de la classe wx.MenuBar ***Append()***.

Cette méthode ne prend que deux paramètres, la référence de l'objet wx.Menu à ajouter, et le titre de ce menu sous forme d'une chaîne de caractères.

Ligne 16

Il ne nous reste plus qu'à lier la barre de menu à notre fenêtre à l'aide de la méthode de la

classe wx.Frame ***SetMenuBar***() qui ne prend pour seul paramètre que la référence de l'objet wx.MenuBar.

A noter que si cette méthode est omise, la barre de menu ne sera pas affichée, bien que créée.

Mais pour qu'une barre de menus soit réellement utile, encore faut-il que lorsqu'on choisit un menu quelque chose se passe. C'est ce que nous allons voir au chapitre suivant...

Lier les éléments d'un menu à des méthodes

Pour lier les actions désirées aux choix dans les menus, il suffit d'intercepter les évènements de la classe wx.CommandEvent à l'aide de la macro wx.EVT_MENU (la classe wx.MenuEvent sert à intercepter des évènements particuliers n'ayant rien avoir avec les choix opérés dans les menus).

Ci-dessous notre exemple précédent complété de gestionnaires d'évènements, et d'une barre d'état chargée de matérialiser ces évènements.

Fichier : Exemple9.py

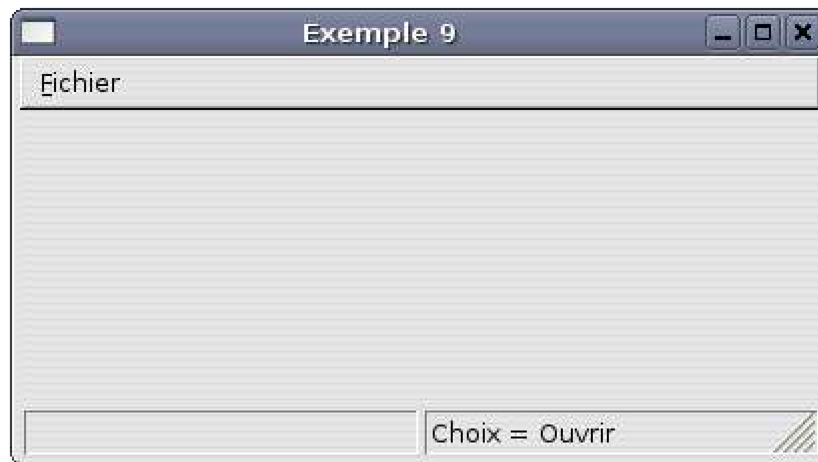
```
01 #!/usr/bin/python
02 # -*- coding: iso-8859-1 -*-
03
04 import wx
05
06 class LesMenus(wx.Frame):
07   def __init__(self, titre):
08     wx.Frame.__init__(self, None, -1, title = titre, size = (400, 200))
09
10     menuFichier = wx.Menu(style = wx.MENU_TEAROFF)
11     menuFichier.Append(wx.ID_OPEN, "&Ouvrir\tCTRL+o", "menu ouvrir")
12     menuFichier.Append(wx.ID_CLOSE, "&Fermer\tCTRL+f", "menu fermer")
13     menuFichier.AppendSeparator()
14     menuFichier.Append(wx.ID_EXIT, "&Quitter\tCTRL+q", "menu quitter")
15
16     menuBarre = wx.MenuBar()
17     menuBarre.Append(menuFichier, "&Fichier")
18     self.SetMenuBar(menuBarre)
19
20     self.barre = wx.StatusBar(self, -1)
21     self.barre.SetFieldsCount(2)
22     self.barre.SetStatusWidths([-1, -1])
23     self.SetStatusBar(self.barre)
24
25     wx.EVT_MENU(self, wx.ID_EXIT, self.OnExit)
26     wx.EVT_MENU(self, wx.ID_OPEN, self.OnOpen)
27     wx.EVT_MENU(self, wx.ID_CLOSE, self.OnClose)
28
29   def OnOpen(self, evt):
30     self.barre.SetStatusText("Choix = Ouvrir", 1)
31
32   def OnClose(self, evt):
33     self.barre.SetStatusText("Choix = Fermer", 1)
34
35   def OnExit(self, evt):
36     self.Destroy()
37
38 class MonApp(wx.App):
39   def OnInit(self):
40     fen = LesMenus("Exemple 9")
41     fen.Show(True)
42     self.SetTopWindow(fen)
43     return True
44
```

```
45 app = MonApp()
46 app.MainLoop()
```

En exécutant ce code, on voit que la partie gauche de la barre d'état située au bas de la fenêtre fait apparaître un message à chaque fois que le curseur de la souris passe sur un élément du menu, et que la partie droite ne fait apparaître un message que quand on clique sur un élément.

Quand on clique sur Quitter, l'application s'arrête.

La capture d'écran qui suit montre l'état de la fenêtre après qu'on ait cliqué sur Ouvrir...



Voyons comment on s'y est pris:

Lignes 11, 12 et 14

On a ajouté à la méthode Append() de wx.Menu le paramètre helpstr qui a pour effet d'en afficher automatiquement le contenu dans la première cellule de la barre d'état de la fenêtre.

Ligne 20

On crée la barre d'état à partir de la classe wx.StatusBar. Si on va consulter sa documentation en ligne, on peut examiner son constructeur :

wxStatusBar(**wxWindow*** *parent*, **wxWindowID** *id*, **long** *style* = *wxST_SIZEGRIP*, **const wxString&** *name* = "*statusBar*")

ce qui transposé en wxPython donne :

wx.StatusBar(*parent*, *id*, *style* = *wxST_SIZEGRIP*, *name* = "*statusBar*")

Seuls les deux premier paramètres sont en règle générale utilisés.

- **Parent** est le conteneur de la barre d'état, ici ***self*** pour notre fenêtre.
- **id** est alimenté à -1 pour laisser wxPython faire son choix tout seul.

Ligne 21

On utilise la méthode ***SetFieldsCount()*** appartenant à la classe ***wx.StatusBar*** pour diviser notre barre d'état en deux zones. Le paramètre attendu par cette méthode est un nombre représentant le nombre de zones voulues.

Ligne 22

Avec la méthode ***SetStatusWidths()*** appartenant elle aussi à la classe ***wx.StatusBar***, on dimensionne les deux zones créées à la ligne précédente.

Son paramètre est une liste python d'autant de valeurs de largeur qu'il y a de zones dans la barre d'état.

Les valeurs positives définissent des zones fixes dont la largeur est exprimée en pixels, et les valeurs négatives représentent des largeurs variables. La valeur des nombres négatifs exprime une proportion entre les largeurs des zones variables qui sont calculées à partir de la somme des largeurs de toutes les zones, déduction faite de la somme des largeurs des zones fixes.

Par exemple, une barre d'état comporte trois zones dont une fixe et deux variables. La zone fixe est dimensionnée à 100 pixels et doit se situer à la droite de la barre d'état, la zone variable la plus à gauche doit être deux fois moins large que la zone variable du centre.

La liste à passer en paramètre de la méthode sera : **[-1, -2, 100]**

Dans le cas présent, on veut deux zones égales, qui feront toujours la moitié de la barre d'état, quelque soit la largeur de celle-ci. On lui a donc passé la liste suivante : **[-1, -1]**.

Ligne 23

De la même façon qu'on a tout à l'heure lié la barre de menu à la fenêtre, on doit faire de même pour la barre d'état. On dispose pour cela de la méthode ***SetStatusBar()*** appartenant à la classe ***wx.Frame***.

Lignes 25 à 27

On intercepte les évènements liés aux choix dans les menus à l'aide de la macro-commande wx.EVT_MENU(), qui prend trois paramètres :

- La classe parente de l'objet wx.Menu, qui est notre wx.Frame. On met donc le mot clé ***self***.
- L'id de l'élément de menu concerné.
- La méthode de classe à laquelle l'évènement doit être passé pour qu'on puise le gérer.

Lignes 29 à 36

On définit les méthodes associées aux événements. Dans les méthodes liées aux éléments de menu «Ouvrir» et «Fermer», on affiche un message dans la barre d'état à l'aide de la méthode ***SetStatusText***() qui prend en paramètres la phrase à afficher, et l'index de la zone à alimenter (l'indexation commence par la gauche avec la valeur 0).

La méthode OnExit() provoque la destruction de la fenêtre et l'arrêt du programme.

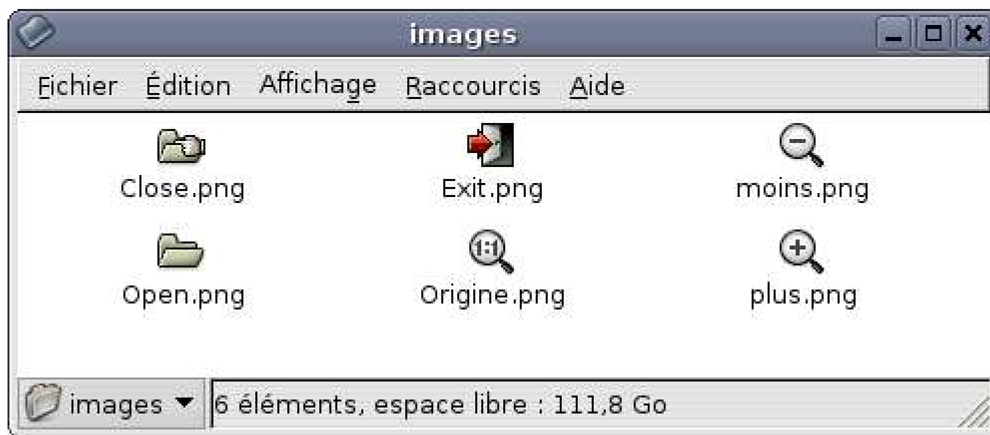
Construisons une véritable application

Maintenant que vous possédez les bases de wxPython, on va tenter de construire une application complète (pas trop complexe), mettant en oeuvre toutes les notions vues jusqu'à présent qu'on va pouvoir compléter de quelques éléments intéressants.

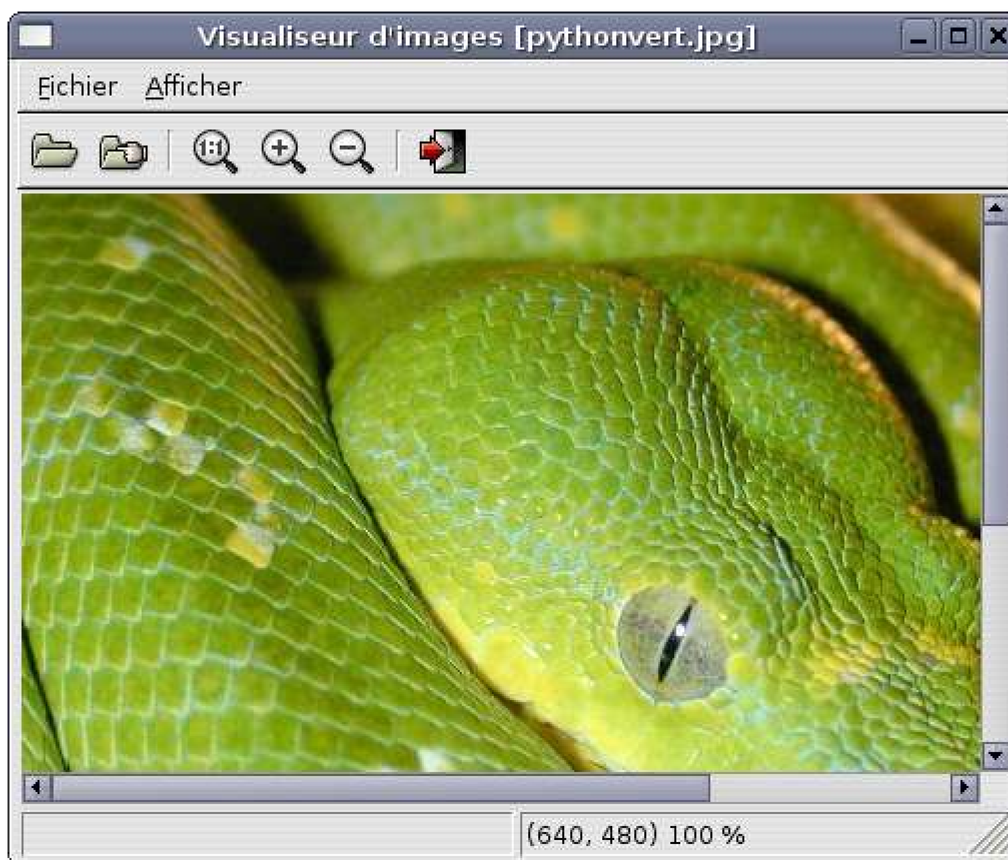
L'application que je vous propose de développer, est un visualiseur d'images de tous formats, avec des fonctions de zoom avant et arrière.

Pour construire la barre d'outils, on va avoir besoin de petites images destinées à décorer les boutons :

J'ai utilisé des images png 24*24 comme celles-ci :



Ci-dessous vous trouverez une copie d'écran de l'application telle qu'elle devrait être après qu'on y ait chargé la photo d'un magnifique python vert.



Code de l'application :

```
001 #!/usr/bin/python
002 # -*- coding: iso-8859-15 -*-
003
004 import wx
005
006 ID_PLUS = 100
007 ID_MOINS = 101
008
009 phrase1 = u"Ramener l'image à sa taille d'origine"
010 phrase2 = u"Opération interdite"
```

```
011 if not wx.USE_UNICODE:
012    phrase1 = phrase1.encode("iso8859-15", "replace")
013    phrase1 = phrase1.encode("iso8859-15", "replace")
014
015 class Visu(wx.ScrolledWindow):
016     def __init__(self, conteneur):
017          wx.ScrolledWindow.__init__(self,parent = conteneur)
018
019
020          self.bmp = None
021          self.image = None
022
023
024     def Affiche(self, bmp, ratio):
025          if self.bmp != None:
026                 posX, posY = self.GetViewStart()
027                 self.image.Destroy()
028                 self.SetScrollRate(0, 0)
029          else:
030                 posX = 0
031                 posY = 0
032          self.bmp = bmp
033          self.SetVirtualSize(wx.Size(bmp.GetWidth(), bmp.GetHeight()))
034          self.image = wx.StaticBitmap(self, -1, self.bmp)
035          self.SetScrollRate((10*ratio)/100, (10*ratio)/100)
036          self.Scroll(posX, posY)
037          self.Refresh()
038
039    def Efface(self):
040          self.image.Destroy()
041          self.bmp = None
042          self.SetScrollRate(0, 0)
043
044 class Principale(wx.Frame):
045    def __init__(self, titre):
046          wx.Frame.__init__(self, None, -1, title = titre, size = (499, 399))
047
048          self.imgORIG = None
049          self.imgORIX = 0
050          self.imgORIY = 0
051          self.bmpRESU = None
052          self.ratio = 100
053          self.inc = 5
054
055
056          menuFichier = wx.Menu(style = wx.MENU_TEAROFF)
057          menuFichier.Append(wx.ID_OPEN, "&Ouvrir\tCTRL+o", "Ouvrir un fichier image")
058          menuFichier.Append(wx.ID_CLOSE,
059                              "&Fermer\tCTRL+f", "Fermer le fichier ouvert")
060          menuFichier.AppendSeparator()
061          menuFichier.Append(wx.ID_EXIT, "&Quitter\tCTRL+q", "Quitter l'application")
062
063          menuAfficher = wx.Menu(style = wx.MENU_TEAROFF)
064          menuAfficher.Append(wx.ID_UNDO,
065                              "&Taille d'origine\tCTRL+t",
066                              phrase1)
067          menuAfficher.Append(ID_PLUS, "&Agrandir\tCTRL+a", "Agrandir l'image")
068          menuAfficher.Append(ID_MOINS, "&Diminuer\tCTRL+a", "Diminuer l'image")
069
070          menuBarre = wx.MenuBar()
071          menuBarre.Append(menuFichier, "&Fichier")
072          menuBarre.Append(menuAfficher, "&Afficher")
073          self.SetMenuBar(menuBarre)
074
075          self.barre = wx.StatusBar(self, -1)
076          self.barre.SetFieldsCount(2)
```

```
077         self.barre.SetStatusWidths([-1, -1])
078         self.SetStatusBar(self.barre)
079
080         outils = wx.ToolBar(self, -1,
081                             style = wx.TB_HORIZONTAL | wx.NO_BORDER)
082         outils.AddSimpleTool(wx.ID_OPEN,
083                             wx.Bitmap("Open.png", wx.BITMAP_TYPE_PNG),
084                             shortHelpString = "Ouvrir",
085                             longHelpString = "Ouvrir un fichier image")
086         outils.AddSimpleTool (wx.ID_CLOSE,
087                             wx.Bitmap("Close.png", wx.BITMAP_TYPE_PNG),
088                             shortHelpString = "Fermer",
089                             longHelpString = "Fermer le fichier ouvert")
090         outils.AddSeparator()
091         outils.AddSimpleTool(wx.ID_UNDO,
092                              wx.Bitmap("Origine.png", wx.BITMAP_TYPE_PNG),
093                              shortHelpString = "Taille originale",
094                              longHelpString = phrase1)
095         outils.AddSimpleTool(ID_PLUS,
096                             wx.Bitmap("plus.png", wx.BITMAP_TYPE_PNG),
097                             shortHelpString = "Agrandir",
098                             longHelpString = "Agrandir l'image")
099         outils.AddSimpleTool(ID_MOINS,
100                             wx.Bitmap("moins.png", wx.BITMAP_TYPE_PNG),
101                             shortHelpString = "Diminuer",
102                             longHelpString = "Diminuer l'image")
103         outils.AddSeparator()
104         outils.AddSimpleTool(wx.ID_EXIT,
105                             wx.Bitmap("Exit.png", wx.BITMAP_TYPE_PNG),
106                             shortHelpString = "Quitter",
107                             longHelpString = "Quitter l'application")
108         outils.Realize()
109         self.SetToolBar(outils)
110
111         sizer = wx.BoxSizer()
112         self.panneau = Visu(self)
113         sizer.Add(self.panneau, 1, wx.EXPAND|wx.ALL, 2)
114         self.SetSizer(sizer)
115
116         wx.EVT_MENU(self, wx.ID_OPEN, self.OnOpen)
117         wx.EVT_MENU(self, wx.ID_CLOSE, self.OnClose)
118         wx.EVT_MENU(self, wx.ID_EXIT, self.OnExit)
119         wx.EVT_MENU(self, wx.ID_UNDO, self.Retour)
120         wx.EVT_MENU(self, ID_PLUS, self.Plus)
121         wx.EVT_MENU(self, ID_MOINS, self.Moins)
122
123     def Retour(self, evt):
124         if self.imgORIG != None:
125             self.ratio = 100
126             self.bmpRESU = self.imgORIG.ConvertToBitmap()
127             self.panneau.Affiche(self.bmpRESU, self.ratio)
128             self.barre.SetStatusText("(%s, %s) %s %%"%(self.imgORIX,
129                                      self.imgORIY, self.ratio), 1)
130
131     def Plus(s1elf, evt):
132         if self.imgORIG != None:
133             self.ratio = self.ratio + self.inc
134             largeur = (self.imgORIX * self.ratio)/100
135             hauteur = (self.imgORIY * self.ratio)/100
136             self.bmpRESU = self.imgORIG.Scale(largeur, hauteur).ConvertToBitmap()
137             self.panneau.Affiche(self.bmpRESU, self.ratio)
138             self.barre.SetStatusText("(%s, %s) %s %%"%(self.imgORIX,
139                                       self.imgORIY, self.ratio), 1)
140
141     def Moins(self, evt):
142         if self.ratio > 5 and self.imgORIG != None:
```

```
143                 self.ratio = self.ratio - self.inc
144                 largeur = (self.imgORIX * self.ratio)/100
145                 hauteur = (self.imgORIY * self.ratio)/100
146                 self.bmpRESU = self.imgORIG.Scale(largeur, hauteur).ConvertToBitmap()
147                 self.panneau.Affiche(self.bmpRESU, self.ratio)
148                 self.barre.SetStatusText("(%s, %s) %s %%"%(self.imgORIX,
149                                           self.imgORIY, self.ratio), 1)
150
151     def OnOpen(self, evt):
152         if self.imgORIG != None :
153             dlg = wx.MessageDialog(self,
154                   "Vous devez d'abord fermer l'image en cours d'utilisation",
155                     phrase2, style = wx.OK)
156             retour = dlg.ShowModal()
157             dlg.Destroy()
158         else:
159             dlg = wx.FileDialog(self, "Choisissez un fichier",
160                                 wildcard = "*.*",
161                                 style = wx.OPEN)
162             retour = dlg.ShowModal()
163             chemin = dlg.GetPath()
164             fichier = dlg.GetFilename()
165             dlg.Destroy()
166             if retour == wx.ID_OK and fichier != "":
167                 self.imgORIG = wx.Image(chemin, wx.BITMAP_TYPE_ANY)
168                 self.imgORIX = self.imgORIG.GetWidth()
169                 self.imgORIY = self.imgORIG.GetHeight()
170                 self.bmpRESU = self.imgORIG.ConvertToBitmap()
171                 self.panneau.Affiche(self.bmpRESU, self.ratio)
172                 self.SetTitle("Visualiseur d'images [%s]"% fichier)
172                 self.barre.SetStatusText("(%s, %s) %s %%"%(self.imgORIX,
174                                           self.imgORIY, self.ratio), 1)
175
176     def OnClose(self, evt):
177         if self.imgORIG != None :
178             self.panneau.Efface()
179             self.imgORIG = None
180             self.imgORIX = 0
181             self.imgORIY = 0
182             self.bmpRESU = None
183             self.ratio = 100
184             self.SetTitle("Visualiseur d'images")
185             self.barre.SetStatusText("", 1)
186
187     def OnExit(self, evt):
188         self.Destroy()
189
190 class MonApp(wx.App):
191     def OnInit(self):
192         wx.InitAllImageHandlers()
193         fen = Principale("Visualiseur d'images")
194         fen.Show(True)
195         self.SetTopWindow(fen)
196         return True
197
198 app = MonApp()
199 app.MainLoop()
```

Les variables globales

Lignes 6 et 7

On définit des ID pour les menus zoom + et zoom – car il n'y a pas de constantes prédéfinies en wxPython pour ces actions.

Lignes 9 à 13

Je travaille sur une version unicode de wxPython, alors que celle qui est installée sur mon Windows XP de test est une version non-unicode.

Si les menus et divers messages de mon application ne comportaient pas de caractères accentués, on ne verrait pas de différence. Mais pour que l'application tourne sans erreur dans les deux environnements, il faut traiter les chaînes de caractères comportant des accentués.

Aux lignes 9 et 10 je déclare les chaînes en tant que chaînes unicodes (avec le u minuscule en préfixe).

A la ligne 11, je teste si la version de wx.Python est une version unicode en récupérant la valeur booléenne de la constante wx.USE_UNICODE, et dans le cas contraire, aux lignes 12 et 13 je décode les chaînes unicode de façon à les mettre au format latin 2 (iso-8859-15).

Construction de la fenêtre principale

C'est la classe Principale() dont l'initialisation est définie entre la ligne 44 et la ligne 121.

Aux lignes 48 à 53, on déclare plusieurs membres de classe qui nous serviront dans la gestion de l'affichage des images (voir la suite).

De la ligne 56 à la ligne 78 on crée le menu et la barre d'état comme on l'a vu dans les exemples précédents.

La barre d'outils

On ajoute la barre d'outils entre la ligne 80 et la ligne 109.

Ligne 80

Pour créer la barre d'outil, il faut d'abord construire le corps de la barre en créant une instance de la classe **wx.ToolBar.**

Si on consulte la documentation on voit que le constructeur de cette classe est en C++ le suivant:

wxToolBar(**wxWindow*** *parent*, **wxWindowID** *id*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxTB_HORIZONTAL | wxNO_BORDER*, **const wxString&** *name = wxPanelNameStr*)

en wxPython ça donne ceci :

instance = **wx.ToolBar**(*parent*, *id*, *pos = wx.DefaultPosition*, *size = wxDefaultSize*, *style = wx.TB_HORIZONTAL | wx.NO_BORDER*, *name = wx.PanelNameStr*)

Vous commencez à avoir l'habitude des paramètres, et il n'y a là rien de bien nouveau par rapport à ce qu'on a vu jusqu'à maintenant.

On crée notre barre d'outil avec les options par défaut.

Lignes 82 à 107

On ajoute les boutons à la barre d'outil à l'aide de sa méthode de classe **AddSimpleTool()**.

Ne cherchez pas dans la documentation en ligne, vous ne trouverez pas cette méthode.

Par contre, si vous utilisez la méthode **AddTool()** qui est décrite dans la documentation, vous aurez systématiquement une erreur au chargement de votre programme.

Confronté à ce problème, c'est dans les démos fournies avec wxPython que j'ai trouvé la méthode utilisée.

Si vous consultez la documentation spécifique de wxPython à l'adresse suivante, http://www.wxpython.org/docs/api , vous constaterez que les méthodes implémentées dans la version wxPython du wxWidget ToolBar sont complètement différentes des méthodes

originales. C'est particulièrement dommageable, surtout que les spécificités wxPython n'ont pas été notées dans la documentation comme cela était habituellement fait.

Mais c'est comme ça.

Donc nous utilisons la méthode **AddSimpleTool()** qui accepte cinq paramètres :

- **id** : ce paramètre est celui qui permettra de relier le bouton à l'évènement à déclencher, comme pour les éléments de menu.

  Si vous observez bien, le programme, vous verrez que j'ai créé, pour chaque élément de menu, sa correspondance sous forme d'un bouton dans la barre d'outil. Et si vous y regardez de plus près, vous constaterez que chaque élément de menu, et sa correspondance en bouton, portent le même Id.

  C'est une particularité de la barre d'outils qui accepte pour ses éléments le même id que les éléments qui leur correspondent dans la barre de menus. L'action sur un bouton utilise donc le gestionnaire d'évènements utilisé pour le menu, et on ne doit donc pas se préoccuper de programmer une gestion séparée.

- **bitmap1** : ce paramètre accepte un objet de type **wx.Bitmap** qui représente l'image affichée sur le bouton.

  On voit dans la documentation que spécifiquement pour wxPython le constructeur C++ du **wx.Bitmap** a été implémenté selon cinq prototypes différents :

  wx.Bitmap(name, flag) charge une image bitmap depuis un fichier

  wxEmptyBitmap(width, height, depth = -1) crée une image bitmap vide

  wxBitmapFromXPMData(listOfStrings) crée une image bitmap depuis une liste Python de chaînes de caractères qui contiennent des données de format XPM

  wxBitmapFromBits(bits, width, height, depth=-1) crée une image bitmap depuis un tableau de bits contenu dans une chaîne

  wxBitmapFromImage(image, depth=-1) Convertit un objet de type **wx.Image** en un objet de type **wx.Bitmap**.

  On va utiliser donc charger nos fichiers png en précisant qu'il s'agit d'images png à l'aide du drapeau wx.BITMAP_TYPE_PNG en utilisant la forme suivante :

  wx.Bitmap(chemin_de_l_image, wx.BITMAP_TYPE_PNG)

- **bitmap2** : Il s'agit de l'image sous forme d'un objet wx.Bitmap qu'on veut voir apparaître quand le bouton est enfoncé.

  Si on ne renseigne pas cet argument, c'est l'image du précédent argument qui est utilisé.

- **shortHelpString** : C'est le texte qu'on veut voir apparaître en popup quand on passe le curseur de la souris sur le bouton.

- **longHelpString** : C'est le texte qu'on veut voir apparaître dans la barre d'état quand on passe le curseur de la souris sur le bouton.

Pour séparer les boutons par groupes, on utilise la méthode de classe **AddSeparator()**, qui positionne une barre de séparation verticale.

Ligne 108

La barre d'outil est constituée complètement lorsque sa méthode **Realize()** sans argument est appelée.

Ligne 109

On attache la barre d'outil à notre fenêtre principale par la méthode **wx.SetToolBar()** dérivée de la classe **wx.Window**.

Le corps de la fenêtre

Il est dessiné entre les lignes 111 et 114.

On crée d'abord une boite de placement de classe **wx.sizer**.

Ensuite on crée une instance de la classe **Visu**, qui est une classe définie dans notre programme, et qui dérive de la classe **wx.ScrolledWindow**. On détaillera plus loin la classe **Visu** et son utilité.

On place ensuite l'objet de type **Visu** dans la boite de placement à l'aide de la méthode **Add()** dérivée de la classe **wx.Sizer**, et enfin on relie Le sizer à la fenêtre à l'aide de la méthode **SetSizer()** dérivée de la classe **wx.Window**.

De la ligne 116 à la ligne 121, on crée les gestionnaires d'évènement des menus à l'aide de la macro **wx.EVT_MENU**.

Ces gestionnaires seront également utilisés par les boutons de la barre d'outils.

La classe Visu

Elle est définie entre les lignes 15 et 42.

Cette classe, qu'on fait dériver de **wx.ScrolledWindow** est destinée à l'affichage des images qu'on va charger dans l'application.

Le seul paramètre attendu, est son conteneur, c'est à dire sa fenêtre parente.

La classe wx.ScrolledWindow

C'est une classe qui dérive de la classe wx.Window, et qui comprend une gestion automatique d'ascenseurs verticaux et horizontaux, dès lors que la taille de son contenu dépasse la taille de sa zone cliente.

Elle gère donc une zone virtuelle avec des coordonnées propres.

C'est l'objet idéal pour afficher des contenus qu'on veut pouvoir agrandir à souhait sans en altérer les proportions.

Le constructeur C++ de la classe **wx.ScrolledWindow** est le suivant :

wxScrolledWindow(**wxWindow*** *parent*, **wxWindowID** *id = -1*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxHSCROLL* | *wxVSCROLL*, **const wxString&** *name* = "*scrolledWindow*")

qu'on peut traduire en wxPython :

wx.ScrolledWindow(*parent*, *id = -1*, *pos = wx.DefaultPosition*, *size = wx.DefaultSize*, *style = wx.HSCROLL* | *wx.VSCROLL*, *name* = "*scrolledWindow*")

Le seul argument obligatoire est l'argument parent, c'est pourquoi c'est aussi le seul argument exigé par la classe **Visu**.

Le style par défaut (wx.HSCROLL | wx.VSROLL) affiche les ascenseurs dans les deux sens, ce qui nous convient.

Dans la méthode __init__(), on ne définit aucun contrôle. On crée deux membres de classe self.bmp et self.image dont on aura utilité dans les méthode de la classe.

La méthode Affiche()

Cette méthode est celle qui affiche l'image chargée par le programme principal.

Elle prend deux paramètres:

- **bmp** : c'est un objet wx.Bitmap qui contient l'image à afficher.
- **ratio** : c'est le ratio utilisé pour calculer la taille de l'image à afficher par rapport à l'image initialement chargée. En fait c'est la proportion de zoom de l'image.

Lignes 25 à 31

On teste si une image est déjà chargée dans le contrôle en évaluant le membre de classe self.bmp.

Si self.bmp existe, on récupère les positions de scrolling des deux ascenseurs dans les variables posX et posY à l'aide de la méthode **GetViewStart()** dérivée de **wx.ScrolledWindow** qui renvoie ces positions sous la forme d'un tuple de deux entiers longs.

Ces positions nous seront utiles pour laisser l'image dans la même position d'affichage en cas de zoom.

On détruit le contrôle **wx.StaticBitmap** déjà présent

Puis on utilise la méthode **SetScrollRate()** dérivée **wx.ScrolledWindow** pour mettre les positions verticales et horizontales à zéro, ce qui, lorsqu'aucun contenu n'est plus présent dans l'objet a pour effet de faire disparaître les ascenseurs. Si cette méthode n'était pas appelée, les ascenseurs seraient toujours présents, même si la nouvelle image chargée était plus petite que la zone cliente de la fenêtre.

Sinon, on valide les positions de scroll à zéro en alimentant posX et posY avec cette valeur.

Lignes 32 à 37

ligne 32, on alimente le membre de classe self.bmp de la référence du **wx.Bitmap** passée en paramètre.

Ligne 33, on utilise la méthode **SetVirtualSize()** dérivée de **wx.ScrolledWindow** pour définir la taille de la zone virtuelle scrollable. On lui donne les valeurs de taille de l'image contenu dans le **wx.Bitmap**, avec les méthodes **GetWidth()** et **GetHeight()** qui en dérivent.

ligne 34, on crée un contrôle **wx.StaticBitmap**.

Le contrôle **wx.StaticBitmap** est un contrôle qui affiche une image contenue dans un objet **wx.Bitmap**. son constructeur prend trois paramètres obligatoires:

- parent : c'est l'id de son conteneur
- id : c'est son propre id (-1 pour un calcul automatique)
- label : c'est un objet **wx.Bitmap** qui contient l'image à y afficher.

ligne 35, on règle l'intervalle de scrolling des ascenseurs avec la méthode SetScrollRate() dérivée de wx.ScrolledWindow.

Cette méthode permet de régler en pixels le déplacement de chaque action de scrolling. Avec comme premier paramètre le nombre de pixels pour l'ascenseur horizontal, et en deuxième paramètre le nombre de pixels pour l'ascenseur vertical.

Appliquer le ratio de dimensionnement de l'image originale sur ces valeurs, permet, en combinaison avec la position des ascenseurs avant redimensionnement, de conserver la position de l'image dans la fenêtre même après un zoom.

ligne 36, on remet les ascenseurs dans la position précédent l'affichage.

ligne 37, on rafraîchit la fenêtre (obligatoire sous MS Windows pour éviter les effets de traînées).

Mise en oeuvre des fonctionnalités

L'ensemble des fonctions du programme se situe au niveau de la classe Principale et de ses méthodes

La méthode OnOpen(self, evt)

Cette méthode est définie entre la ligne151 et la ligne 174 et est appelé quand on clique sur le bouton «ouvrir»ou sur l'élément de menu correspondant.

ligne 152

On teste si la valeur du membre de la classe Principale, imgORIG est différente de None. C'est à dire qu'on s'assure qu'une image n'est pas déjà chargée.

Dans le cas contraire, on lance un message d'avertissement demandant à l'utilisateur de fermer cette image avant d'en charger une autre.

Pour cela, à la ligne 153, on utilise une boite de dialogue construite avec la classe **wx.MessageDialog**, qui dérive de la classe **wx.Dialog** dont le constructeur est définit en c++ de la façon suivante :

wxMessageDialog(**wxWindow*** *parent*, **const wxString&** *message*, **const wxString&** *caption* = *"Message box"*, **long** *style* = *wxOK* | *wxCANCEL*, **const wxPoint&** *pos* = *wxDefaultPosition*)

Ce qui donne en wxPython :

wx.MessageDialog(*parent*, *message*, *caption* = *"Message box"*, *style* = *wx.OK* | *wx.CANCEL*, *pos* = *wx.DefaultPosition*)

parent : La fenêtre parente (en général, celle qui appelle)

message : Le message qu'on veut afficher

caption : le titre de la boite de dialogue

style : une combinaison des constantes suivantes

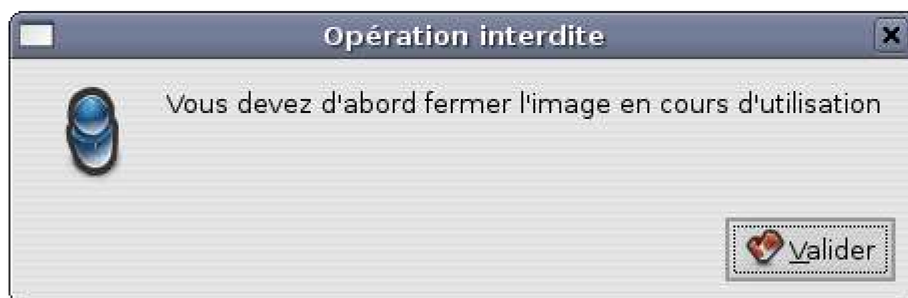
wx.OK	Affiche un bouton Ok
wx.CANCEL	Affiche un bouton Annuler
wx.YES_NO	Affiche les boutons oui et non
wx.YES_DEFAULT	idem **wx.YES_NO**, mais oui possède le focus par défaut.
wx.NO_DEFAULT	idem **wx.YES_NO**, mais non possède le focus par défaut.
wx.ICON_EXCLAMATION	Affiche l'icône point d'exclamation.
wx.ICON_HAND	Affiche l'icône erreur.
wx.ICON_ERROR	idem **wx.ICON_HAND**.
wx.ICON_QUESTION	Affiche l'icône point d'interrogation.
wx.ICON_INFORMATION	Affiche l'icône information (i)
wx.STAY_ON_TOP	Le message reste toujours visible (MS Windows seulement).

pos : la position de la fenêtre sous la forme d'un objet wx.Point

S'agissant d'un message d'avertissement, on va n'afficher que le bouton Ok.

A la ligne 156, on affiche notre boite de dialogue à l'aide de la méthode **ShowModal()** dérivée de la classe **wx.Dialog**, et qui permet d'interdire toute inter-action avec le programme principal tant que la boite de dialogue reste visible.

La fenêtre sur mon système a l'apparence suivante:



Une fois la boite de dialogue fermée (quand l'utilisateur à cliqué sur Ok), on la détruit avec la

méthode **destroy()** (ligne 157), afin qu'elle n'encombre pas la mémoire.

Si aucune image n'est encore chargée dans l'application, on passe à la ligne 159 pour créer une boite de dialogue d'ouverture de fichier.

On utilise pour cela la classe **wx.FileDialog** qui, elle aussi, dérive de **wx.Dialog**.

Son constructeur, en C++ est le suivant :

wxFileDialog(**wxWindow*** *parent*, **const wxString&** *message = "Choose a file"*, **const wxString&** *defaultDir = ""*, **const wxString&** *defaultFile = ""*, **const wxString&** *wildcard = "*.*"*, **long** *style = 0*, **const wxPoint&** *pos = wxDefaultPosition*)

Ce qui donne en wxPython:

wx.FileDialog(*parent*, *message = "Choose a file"*, *defaultDir = ""*, *defaultFile = ""*, *wildcard = "*.*"*, *style = 0*, *pos = wx.DefaultPosition*)

Signification des paramètres

parent *:* La fenêtre parente

message *:* Message à afficher.

defaultDir *:* Le répertoire de recherche par défaut, ou une chaîne vide.

defaultFile *:* Le fichier à rechercher par défaut ou une chaîne vide.

wildcard*:* le prototype tel que "*.*" ou "BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif".

Notez qu'un seul prototype est possible à la fois, et qu'il n'est pas possible d'afficher plus d'un type de fichier à la fois, sauf si le prototype est «*.*»

style *:* Un style parmi les suivants :

wx.OPEN	Pour une boite de dialogue d'ouverture de fichier
wx.SAVE	Pour une boite de dialogue de sauvegarde de fichier
wx.OVERWRITE_PROMPT	Pour sauvegarde uniquement: demande une confirmation pour écraser un fichier existant.
wx.MULTIPLE	Pour ouverture uniquement: permet de sélectionner plusieurs fichiers à la fois
wx.CHANGE_DIR	Change le répertoire par défaut en prenant celui au sein duquel ont été choisis les fichiers.

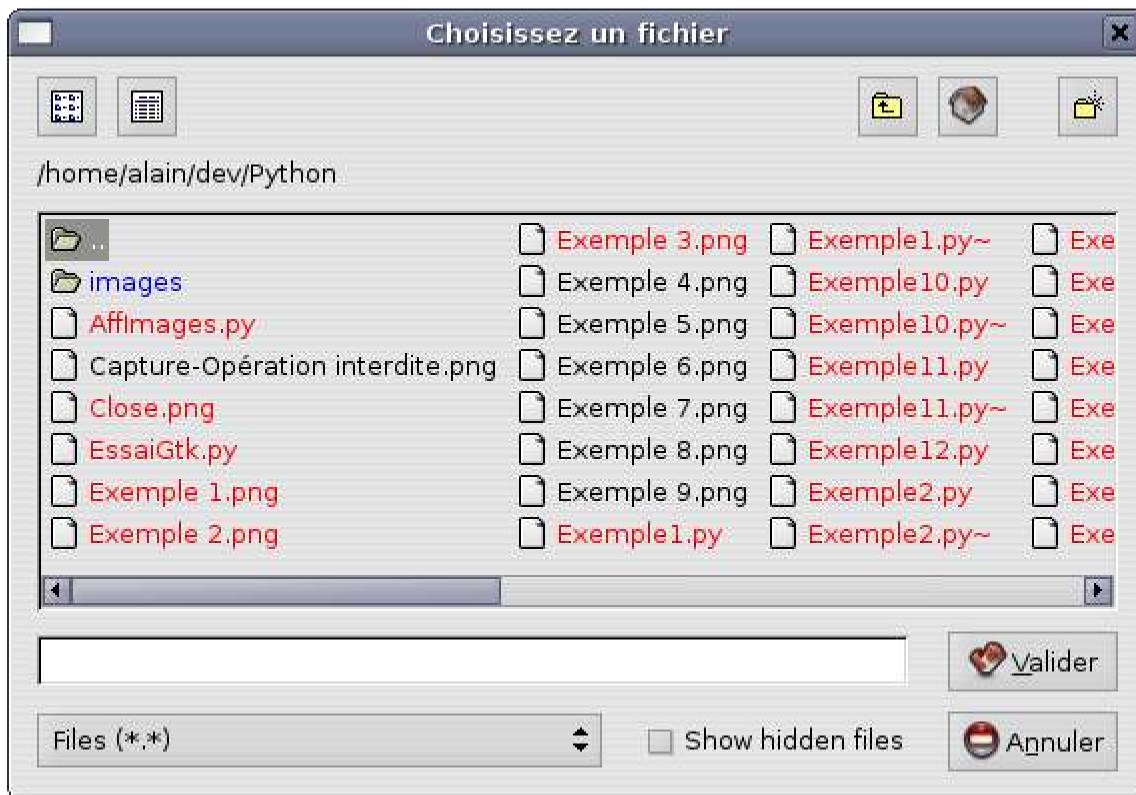
pos : Non implémenté à ce jour

On choisit le style wx.OPEN puisqu'on veut charger un fichier. Le wildcard est positionné à *.* de façon à obtenir l'affichage de tous les types de fichiers simultanément.

A la ligne 162, on affiche la fenêtre à l'aide de la méthode ShowModal() déjà décrite plus haut, et on stocke la valeur de retour dans une variable. Cette valeur varie selon les boutons positionnés à l'aide des styles dans les boites de dialogue. et peut-être une des suivantes:

```
wx.ID_OK
wx.ID_CANCEL
wx.ID_APPLY
wx.ID_YES
wx.ID_NO
```

Cela donne ceci sur mon système :



Tant que la boite de dialogue est affichée, le programme reste en attente.

Ensuite, à la ligne 163 on récupère le chemin complet du fichier choisi à l'aide de la méthode **GetPath()**, qui renvoie un chaîne vide en cas de non choix.

Et à la ligne 164, on récupère le nom du fichier choisi sans son chemin (pour pouvoir l'afficher plus facilement). Là encore, la méthode **GetFilename()** employée renvoie une chaîne vide en cas de non choix.

On détruit la boite de dialogue qui ne nous sert plus à rien à la ligne 165.

Ensuite, à la ligne 166 on teste si l'utilisateur a bien pressé le bouton Ok et si il a bien choisi un fichier.

Si les deux conditions sont remplies, on va charger le fichier image.

Cela se fait à la ligne à la ligne 167 où on stocke dans le membre de classe imgORIG (pour image d'origine) un objet de type **wx.Image** après y avoir chargé le fichier.

Pourquoi **wx.Image** et non pas **wx.Bitmap** comme on l'avait vu pour les images des boutons ?

Tout simplement parce que l'objet **wx.Image** possède des méthodes de manipulation d'image que **wx.Bitmap** ne possède pas.

Si vous consultez la documentation en ligne, vous constaterez que le constructeur de **wx.Image** possède cinq prototypes différents et spécifiques à l'implémentation wxPython.

wx.Image(name, flag)	Charge une image depuis un fichier
wx.NullImage()	Crée une image vide
wx.EmptyImage(width, height)	Crée une image vide avec dimensions précisées
wx.ImageFromMime(name, mimetype	Charge une image depuis un fichier avec le type mime indiqué
wx.ImageFromBitmap(bitmap)	Crée une image à partir d'un bitmap spécifique à la plateforme

C'est le premier prototype qui nous convient, puisque le but du programme est d'afficher le

contenu d'un fichier image.

Le paramètre flag donne le type de fichier à charger. Il peut prendre les valeurs suivantes :

wx.BITMAP_TYPE_BMP	Charge un bitmap MS Windows.
wx.BITMAP_TYPE_GIF	Charge un GIF
wx.BITMAP_TYPE_JPEG	Charge un JPEG.
wx.BITMAP_TYPE_PNG	Charge un PNG.
wx.BITMAP_TYPE_PCX	Charge un PCX.
wx.BITMAP_TYPE_PNM	Charge un PNM.
wx.BITMAP_TYPE_TIF	Charge un TIFF.
wx.BITMAP_TYPE_XPM	Charge un XPM.
wx.BITMAP_TYPE_ICO	Charge un fichier d'icône MS Windows (ICO).
wx.BITMAP_TYPE_CUR	Charge un fichier curseur MS Windows (CUR).
wx.BITMAP_TYPE_ANI	Charge un curseur animé MS Windows (ANI).
wx.BITMAP_TYPE_ANY	Tente de déterminer automatiquement le format.

Comme l'utilisateur peut choisir n'importe quel type d'image, on choisi le drapeau wx.BITMAP_TYPE_ANY de façon à ce que l'objet **wx.Image** s'auto-détermine.

Une fois l'image chargée, on récupère aux lignes 168 et 169 les dimensions de l'image à l'aide des méthodes **GetWidth()** et **GetHeight()** de la classe **wx.Image** .

L'image étant destinée à être affichée dans le **wx.StaticBitmap**() de la classe **Visu**, il est obligatoire d'en créer une copie sous la forme d'un objet **wx.Bitmap**. En effet, un objet **wx.Image** est un objet qui ne peut être affiché en l'état.

C'est ce qu'on fait à la ligne 170 à l'aide de la méthode **ConvertToBitmap**().

En ligne 171, le nouvel objet est alors passé en paramètre à la méthode **Affiche**() de l'instance de la classe **Visu**, ainsi que le ratio (initialisé à 100).

En ligne 172 on ajoute le nom du fichier image au titre de la fenêtre, et en ligne 173 la taille de l'image et le ratio d'affichage dans la zone 2 de la barre d'état.

La méthode Plus(self, evt)

Cette méthode est appelée lorsque l'utilisateur clique sur le bouton Zoom + de la barre d'outils ou le menu agrandir.

A la ligne 132, on s'assure qu'une image est bien chargée.

Ensuite, on modifie la valeur de ratio en lui ajoutant la valeur du membre de classe inc (5%)

On calcule de nouvelles dimensions en appliquant le nouveau ratio aux dimensions originales de l'image.

En ligne 136, on crée un objet **wx.Bitmap** à partir de l'objet **wx.Image** original, retaillée à l'aide de la méthode **Scale**() aux nouvelles dimensions calculées.

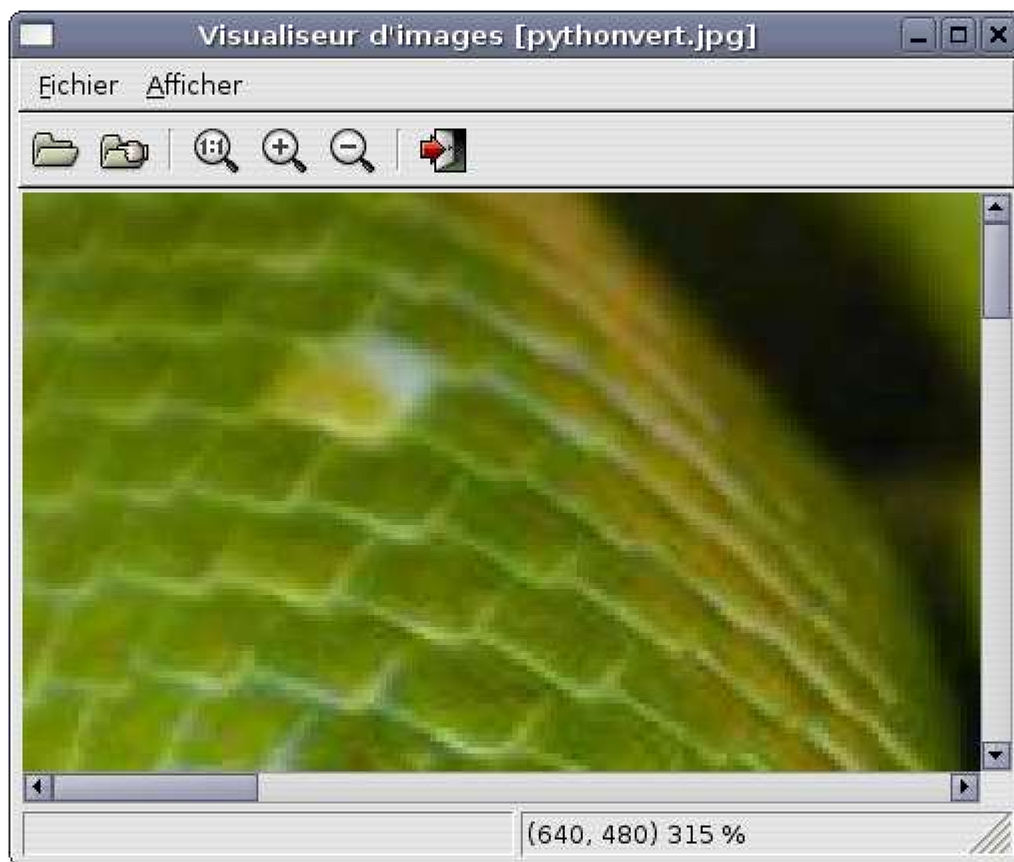
En ligne 137, on appelle la méthode **Affiche**() de l'instance de la classe **Visu**, en lui passant ce nouveau Bitmap, accompagné de la nouvelle valeur de Ratio.

Le **wx.StaticBitmap** contenant l'image avant zoom est alors détruit, pour laisser la place à un nouveau **wx.StaticBitmap** contenant l'image redimensionnée.

On repart systématiquement de l'image originale pour appliquer le nouveau ratio, pour éviter la dégradation de la qualité de l'image affichée qui ne manquerait pas de se produire si on partait de l'image précédemment affichée.

En ligne 137, on met à jour la barre d'état.

Voici ce que donne notre python vert zoomé à 315%:



La méthode Moins(self, evt)

Elle est appelée lorsque l'utilisateur clique sur le bouton zoom- ou le menu diminuer.

C'est exactement le même mécanisme que celui mis en oeuvre dans la méthode Plus() avec décrémentation du ratio au lieu de l'incrémentation.

Simplement, on s'assure à la ligne 142 que le ratio ne descendra pas en dessous de zéro.

Ci-après Notre python vert ramené à 15% de ses dimensions originales.

La méthode Retour(self, evt)

Cette méthode est appelée quand l'utilisateur clique sur le bouton taille originale ou le menu correspondant.

Il s'agit tout simplement de remettre l'image originale en affichage et le ratio à 100%

La méthode OnClose(self, evt)

Cette méthode est appelée quand l'utilisateur clique sur le bouton fermer ou le menu correspondant.

La méthode **Efface**() de l'instance de la classe **Visu** est appelée, qui détruit le **wx.StaticBitmap** et donc l'image contenue.

Tous les membres des deux classes sont remises en l'état initial, de façon à pouvoir charger sans inconvénient une nouvelle image.

La méthode OnExit(self, evt)

Cette méthode est appelée lorsque l'utilisateur clique sur le bouton quitter, le menu correspondant ou clique sur la croix de fermeture de la fenêtre.

La fenêtre est alors détruite ainsi que toutes ses composantes, et la mémoire est libérée. Le toplevel de l'application wxPython étant détruit, l'application s'arrête.

La classe App()

On a vu lors des exemples précédents l'utilité de la classe App, sans laquelle aucune application wxPython ne pourrait s'exécuter.

J'attire simplement votre attention sur la **ligne 192** où la fonction **wx.InitAllImageHandlers()** est appelée. Il s'agit d'une fonction qui initialise les mécanisme de traitement des images de tous types dans wxPython. Sans l'appel à cette fonction en début de programme, nous n'aurions pas pu manipuler les images dans notre programme.

Conclusion

Nous arrivons au terme de ce tutoriel qui n'a pour prétention que de vous permettre d'aborder plus facilement l'utilisation de wxPython pour offrir des interface graphiques de qualité à vos applications Python.

J'espère que ma modeste contribution vous aura été utile, et qu'elle vous encouragera à approfondir un framework dont nous n'avons fait qu'approcher de loin toutes les possibilités et la puissance qu'il vous offre.

Bonne pythonade...

Licence

Copyright © 2005 Alain DELGRANGE. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts. Cette page est déposée à la SACD.