Cours Python II

Franck Delahaye - Observatoire de Paris

Sommaire

1 - Scipy – Quelques exemples d'outils

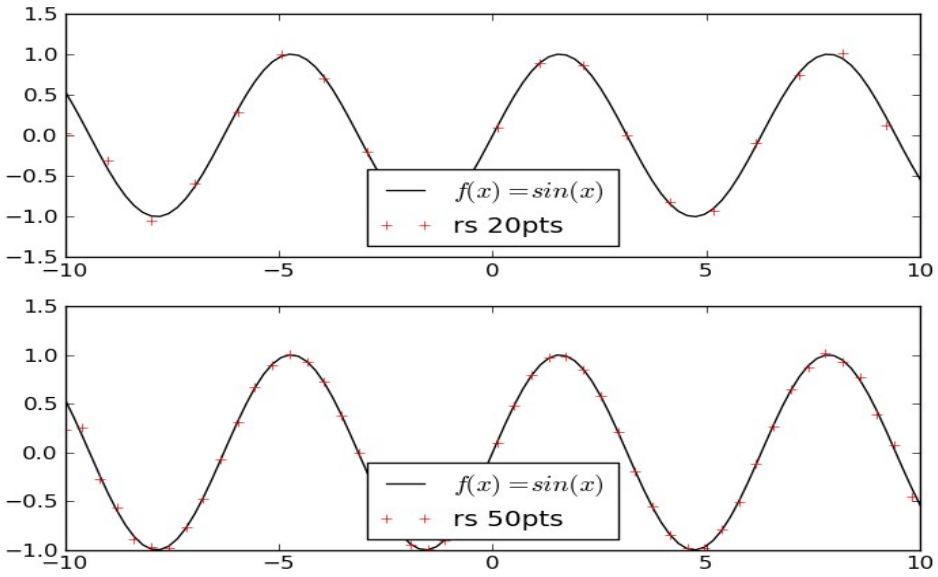
Nous avons évoqué Scipy brièvement dans la partie I du cours pour citer les différents champs d'application (statistiques, optimisation, intégration numérique, algèbre linéaire, transformée de Fourier, traitement du signal, traitement d'images, intégration d'équations différentielles, fonctions spéciales) de ce module. Scipy est construit à partir de Numpy, ce qui signifie qu'il faut avoir le module Numpy pour faire fonctionner le module Scipy. En effet nombre de fonctions ainsi que le type *'ndarray'* de Scipy sont en fait ceux définis dans Numpy. Nous allons voir quelques exemples d'outils de Scipy pour le traitement du signal, les statistiques, l'interpolation numérique et l'intégration numérique.

Note : Contrairement au premier cours nous allons importer individuellement les modules. En effet si *ipython -pylab* permet d'avoir accès aux fonctions principales de numpy, matplolib et scipy, il y a certains sous modules dont nous aurons besoin qui ne sont pas chargés même si ceux ci appartiennent aux différents paquets. De plus cela permettra de repérer quelques uns de ces sous modules et les objets qu'ils comportent.

1.1 - Traitement du signal

Les outils de traitement du signal sont définis dans le module '*scipy.signal*'. Il y a aussi des méthodes associées aux FFT dans le pack '*fftpack*'. On peut ainsi faire des convolutions, du filtrage, déterminer des fonctions de transfert.

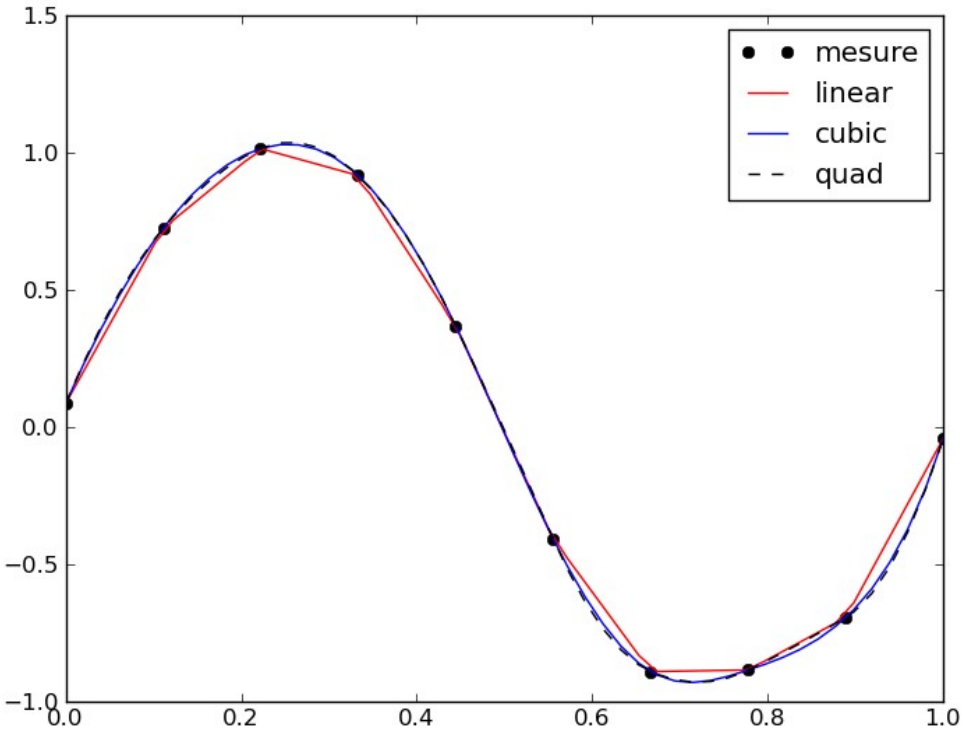
```
import scipy as sp     # import du module scipy. L'appel des fonctions se fera avec le préfixe sp.
import numpy as np  #   ------------------- numpy ---------------------------------------------------- np.
from scipy import signal # sous module signal appartenant à scipy non chargé avec sp (ligne1)
from matplotlib.pyplot import *  import des outils de graphe avec accès direct sans préfixe
x = np.linspace(-10, 10, 100)
ysin = np.sin(x)  # signal sinusoïdale
ysin_rs20 = signal.resample(ysin,20)     # On échantillonne sur 20 points. L'échantillonnage est
ysin_rs50= signal.resample(ysin,50)      # fait en utilisant une FFT. Attention au bord. L'utilisation
                                         # de la FFT suppose une périodicité du signal
```



1.2 - Interpolation numérique

Le module d'interpolation de Scipy comprend plusieurs méthodes d'interpolation définies sous formes de classes. Il est possible d'utiliser des interpolations linéaire, cubique …. Il faut instancier la classe pour l'utiliser.

```
import scipy as sp
import numpy as np
from scipy import interpolate
from matplotlib.pyplot import *

x_measure = np.linspace(0.,1,10)
bruit = np.random.uniform(-0.1,0.1,10)
y_measure = np.sin(2 * np.pi * x_measure) + bruit

# instanciation de la classe interpolation avec diffèrent type d'algorithme – linéaire, cubic,
#quadratic
interp_lin = interpolate.interp1d(x_measure,y_measure)
interp_cubic = interpolate.interp1d(x_measure,y_measure,kind='cubic')
interp_quad = interpolate.interp1d(x_measure,y_measure,kind='quadratic')
#
x_computed = np.linspace(0,1.,50)
y_int_lin = interp_lin(x_computed)
y_int_cub = interp_cubic(x_computed)
y_int_quad = interp_quad(x_computed)
```



1.3 - Intégration numérique

Scipy propose une série de classes pour l'intégration. Cet ensemble se trouve regroupé dans le sous-module '*scipy.integrate*'.

```
In [1]: from scipy import integrate

In [2]: ? integrate
Type:           module
Base Class:     <type 'module'>
String Form:    <module 'scipy.integrate' from '/usr/lib/python2.6/dist-
packages/scipy/integrate/__init__.pyc'>
Namespace:      Interactive
File:           /usr/lib/python2.6/dist-packages/scipy/integrate/__init__.py
Docstring:
    Integration routines
    ====================

    Methods for Integrating Functions given function object.

      quad          -- General purpose integration.
      dblquad       -- General purpose double integration.
      tplquad       -- General purpose triple integration.
      fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
      quadrature    -- Integrate with given tolerance using Gaussian quadrature.
      romberg       -- Integrate func using Romberg integration.

    Methods for Integrating Functions given fixed samples.

      trapz         -- Use trapezoidal rule to compute integral from samples.
      cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
      simps         -- Use Simpson's rule to compute integral from samples.
      romb          -- Use Romberg Integration to compute integral from
                    (2**k + 1) evenly-spaced samples.

      See the special module's orthogonal polynomials (special) for Gaussian
        quadrature roots and weights for other weighting factors and regions.

    Interface to numerical integrators of ODE systems.

      odeint        -- General integration of ordinary differential equations.
      ode           -- Integrate ODE using VODE and ZVODE routines.
```

On peut intégrer une fonction définie ou bien faire une intégration à partir d'un échantillon de points ou encore résoudre un système d'équations différentielles simple. Voici trois exemples d'intégration.

Intégration d'une fonction.

```
import numpy as np
from scipy import integrate

def maFonction(x):
    return sin(x)

In [4]: integrate.quad(maFonction,0,np.pi)  #  ∫_0^Pi sin(x) dx = −cos(Pi) + cos(0)
Out[4]: (2.0, 2.2204460492503131e-14)   # Résultat de l'intégration et erreur
```

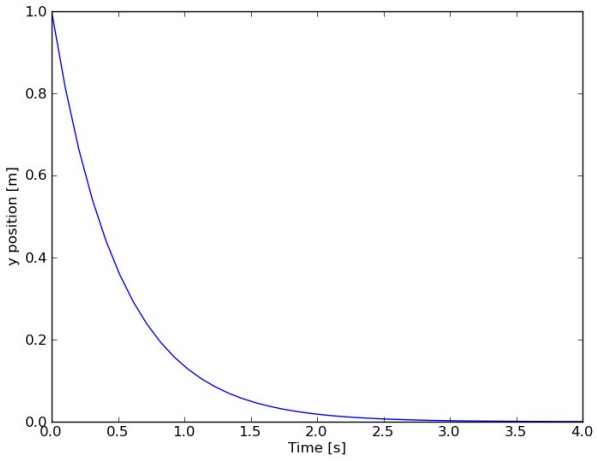
Intégration à partie d'un échantillon de point.

```
import numpy as np
from scipy import integrate

x = linspace(0,np.pi,1000)
y = sin (x)
In [4]: integrate.trapz(y, x, dx = 0.1)  #  ∫_0^Pi sin(x) dx = −cos(Pi) + cos(0)
Out[4]:1.9999983517708524  # Résultat de l'intégration
```

Résolution d'une équation différentielle.

Résoudre l'équation différentielle: dy/dt = -2y entre t = 0 et 4, avec la condition initiale y(t=0) = 1.

```
import numpy as np
from scipy.integrate import odeint

def calc_derivative(ypos, time):
    return -2*ypos

time_vec = np.linspace(0, 4, 40)
yvec = odeint(calc_derivative, 1, time_vec)

import pylab as plt
plt.plot(time_vec, yvec)
plt.xlabel('Time [s]')
plt.ylabel('y position [m]')
```

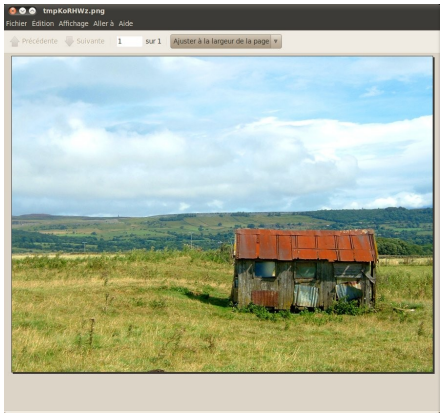


1.4 - Traitement d'images

Les outils de Scipy permettent aussi de manipuler des images. Celles ci sont considérées comme des matrices. Les scripts suivants chargent certains modules (sous modules) individuellement afin de repérer la provenance des différentes fonctions. On peut retrouver la même fonction dans différents modules. Il existe d'autres modules qui permettent certaines manipulations du même genre (matplotlib, Image)

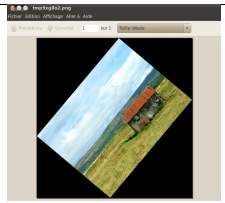
Ouverture et visualisation de l'image

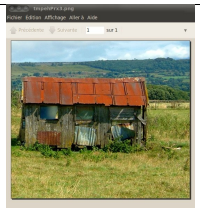
```
In [1]: import scipy.misc as spm
In [2]: fnature = spm.imread("nature.jpg")
In [3]: fnature
Out[3]:
array([[[222, 249, 255],
        [215, 242, 253],
        [208, 235, 246],
        ...,
        [201, 238, 254],
        [199, 238, 253],
        [197, 236, 251]],
In [4]: fnature.size
Out[4]: 5760000
In [5]: fnature.shape
Out[5]: (1200, 1600, 3)
In [6]: bw_nature = spm.imread("nature.jpg",flatten=1)
In [7]: bw_nature.size
Out[7]: 1920000
In [8]: bw_nature.shape
Out[8]: (1200, 1600)
In [9]: spm.imshow(fnature)  # cette commande lance un programme de visualisation d'image
In [10]: spm.imshow(bw_nature)              # présent sur l'ordinateur. (gv, documentViewer …)
```



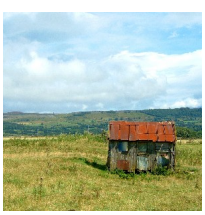
rotation, recadrage, redimensionnement

```
In [7]: import scipy.ndimage as ndi
In [8]: rnature = ndi.rotate(fnature,50)
In [9]: pm.imshow(rnature)
In [10]: crop_nature= fnature[500:-20,800:-30]
In [11]: spm.imshow(crop_nature)
```



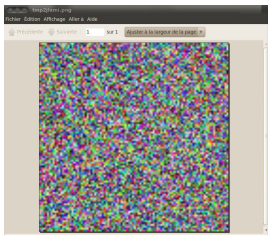


```
In [12]: resize_nature = spm.imresize(nature,(300,400))
In [13]: square_nature = spm.imresize(nature,(300,300))
In [14]: spm.imshow(square_nature)
In [15]: spm.imshow(resize_nature)
```



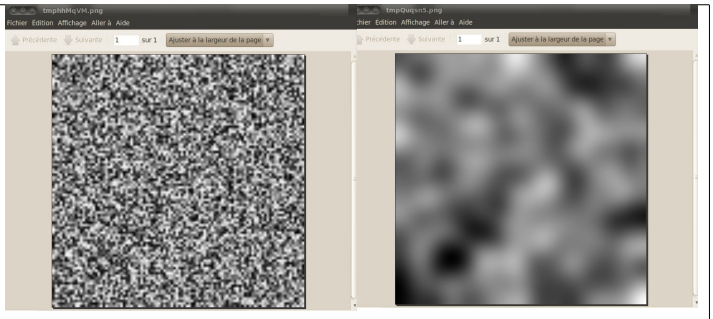
Le format de la matrice dépend du type d'image (couleur ou niveau de gris)

```
In [10]: import scipy as sp
In [11]: a_bw = sp.ones((100,100)) # grayscale
In [12]: for i in range(100):
   ....:         for j in range(100):
   ....:             a_bw[i,j] = sp.random.uniform(0,256)
In [13] : imshow(a_bw)
In [14]: a = sp.ones((100,100,3))  # image couleur RGB
In [15]: for i in range(100):
   ....:         for j in range(100):
   ....:             for k in range(3):
   ....:                 a[i,j,k] = sp.random.uniform(0,256)
In [16] : imshow(a)
```



Il existe des filtres prédéfinis.

```
In [17]: import scipy.ndimage as spi
In [18]: gabw = spi.gaussian_filter(a_bw,5)
In [19]: spm.imshow(gabw)
```



On peut créer des masques ou combiner des images

```
In [17]: y, x = np.ogrid[0:300,0:400] # Indices des coordonnées des pixels d'une image 300x400
In [18]: centre_x, centre_y = 150, 200 # coordonnées du centre de l'image
In [19]: mask = ((y - centery)**2 + (x - centerx)**2)> 150**2 # mask circulaire de rayon 150 pix.
In [53]: res = spm.imresize(bw_nature,(300,400))
In [54]: res[mask]=0 # application du masque
In [55]: spm.imshow(res)
```

Note : Le masque est une matrice de booléens avec 'true'. L'affectation de valeur pour les éléments de matrice de l'image se fait comme pour tout array.

2 - Python et les codes en C et Fortran

Python est développé en C ainsi la plupart des méthodes écrites en C sont optimisées ce qui rend leur exécution rapide. Dans le cadre spécifique de projet on peut être amené à devoir développer des routines qui nécessitent beaucoup de calcul. Python sera certainement moins performant et développer dans son langage compilé habituel peut être une solution préférable. On peut aussi imaginer le cas où toute la création de données de base est faite par des routines déjà existantes dans un langage compilé mais qu'on désire coupler ces routines à un script de post-traitement développé en langage de script.

Python permet d'intégrer des routines fortran et C dans un script. Elles sont chargées comme des modules complémentaires. Pour cela il faut, lors de la compilation, générer un librairie compréhensible pour python.

2.1 - Routine Fortran

Dans le cas de routines fortran, il existe un 'interfaceur' spécifique qui est installé automatiquement lors de l'installation du module Scipy. Cet 'interfaceur' est *f2py (Fortran to Python interface generator)*. Il appelle le compilateur fortran de votre ordinateur f77, f90, gfortran ou ifort pour la compilation et ajoute une enveloppe/un conditionnement pour permettre l'importation dans les scripts python. Voici un exemple.

```
C Fichier Fortran File hello.f
      subroutine myFortranSub (a)
      integer a
      print*, "Hello from Fortran!"
      print*, "a=",a
      end
```

Compilation avec f2py. Cela génère un module appelé hello.so

```
f2py -c -m hello hello.f
```

Utilisation dans python.

```
In [8]: import hello
In [9]: ? hello
Type:           module
Base Class:     <type 'module'>
String Form:    <module 'hello' from 'hello.so'>
Namespace:      Interactive
File:           /home/fdelahaye/WORK/CoursCNAP/Python/Avance/hello.so
Docstring:
   This module 'hello' is auto-generated with f2py (version:2).
   Functions:
     myfortransub(a)
   .
```

```
In [10]: hello.
hello.__class__        hello.__init__         hello.__sizeof__
hello.__delattr__      hello.__name__         hello.__str__
hello.__dict__         hello.__new__          hello.__subclasshook__
hello.__doc__          hello.__package__      hello.__version__
hello.__file__         hello.__reduce__       hello.f
hello.__format__       hello.__reduce_ex__    hello.myfortransub
hello.__getattribute__ hello.__repr__         hello.so
hello.__hash__         hello.__setattr__
In [10]: hello.myfortransub(5)
 This is a fortran subroutine
  a =           5
```

2.2 - Routine C

Pour les routines en C il est existe un 'interfaceur' (wrapper ou créateur d'enrobage) appelé swig. Il est présent dans les paquets standards des distributions linux. On peut aussi le télécharger à l'adresse suivante http://www.swig.org/download.html (pour Linux ou Windows). Ce 'wrapper' n'est pas exclusivement pour Python mais peut aussi générer un interfaçage avec Perl, Java, PHP, Ruby, XML entre autres. Voici la procédure pour inclure les routines C dans un script python.

Pour le programme précédent équivalant nous avons un programme en C, un fichier 'entête' (.h) et un fichier 'information' (.i) suivant.

```
#include <stdio.h>
#include "chello.h"
int croutine(int a)
{
   printf("Hello from C\n");
   printf("Le nombre est %i \n", a);
}
```

Il faut ensuite définir le header (nom_de_fichier.h, ici chello.h)

```
/* File: chello.h */
int croutine(int a);
```

Enfin il faut définir le fichier d'information permettant de construire le '*wrapper*' – nom_fichier.i

```
/* File: chello.i */
%module chello

%{
#define SWIG_FILE_WITH_INIT
#include "chello.h"
 %}
int croutine(int a);
```

La création du module python passe par la création du wrapper, la compilation du code C et le 'linkage' pour la création de la librairie.

```
swig -python chello.i     # création du wrapper haut niveau python
gcc -O2 -fPIC -c chello.c   # compilation
gcc -O2 -fPIC -c  chello_wrap.c -I/usr/include/python2.6  # compilation du wrapper bas niveau C
gcc -shared chello.o chello_wrap.o -o  _chello.so # création de la librairie
```

On peut maintenant accéder au module dans les scripts python.

```
In [1]: import chello
 In [2]: ? chello
Type:           module
Base Class:     <type 'module'>
String Form:    <module 'chello' from 'chello.py'>
Namespace:      Interactive
File:           /home/fdelahaye/WORK/CoursCNAP/Python/Avance/chello.py
Docstring:
   <no docstring>
In [3]: chello.
chello.__builtins__          chello.__str__   chello.__class__          chello.__subclasshook__
chello.__delattr__           chello._chello   chello.__dict__           chello._newclass
chello.__doc__               chello._object   chello.__file__           chello._swig_getattr
chello.__format__            chello._swig_property chello.__getattribute__     chello._swig_repr
chello.__hash__              chello._swig_setattr chello.__init__
chello._swig_setattr_nondynamic
chello.__name__              chello.c chello.__new__              chello.croutine
chello.__package__           chello.h         chello.__reduce__           chello.i
chello.__reduce_ex__         chello.o         chello.__repr__             chello.py
chello.__setattr__           chello.pyc       chello.__sizeof__
In [3]: chello.croutine(5)
Hello from C
Le nombre est 5
In [4]: chello.croutine(6)
Hello from C
Le nombre est 6
```

Pour plus de détails :

http://www.scipy.org/F2py

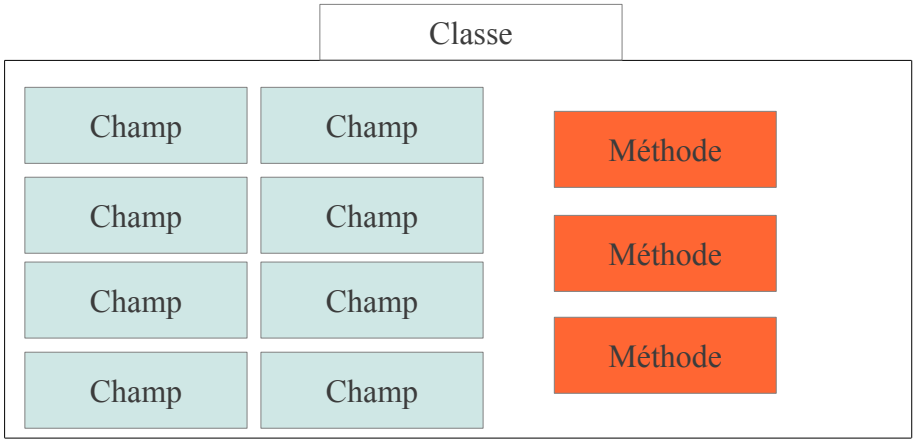
http://cens.ioc.ee/projects/f2py2e/usersguide/index.html

http://www.swig.org

3 - Python et la programmation objet

3.1 - Rappel des principes de base de la Programmation Orientée Objet.

La programmation structurée tend à organiser le code en répondant à la question *'Que doit faire le programme ?'*. Les traitements ou routines occupent le cœur de la programmation. A l'opposé, la programmation orientée objet place les données au centre de l'organisation du code en tentant de répondre à la question *'Sur quoi porte le programme ?'*. Dans cette approche toute entité est un objet. Un objet comprend un groupe de données appelés *attributs* et un ensemble de procédures, appelées *méthodes*, décrivant/opérant la manipulation de l'objet. L'ensemble *Attributs – Méthodes* est regroupé ou défini par une *Classe*. (Les champs ci-dessous correspondent à des attributs modifiables.)



Exemple : Un Article en stock dans un magasin est caractérisé par :

une référence

un prix hors taxe

une quantité .

Pour cet objet on peut définir des méthodes telles que :

PrixTTC = Calculer le prix TTC

PrixRemise = Calculer le prix avec remise

SortieArticle = retirer du stock

EntreeArticle = ajouter au stock

Le fait d'avoir tout regroupé sous une même entité (la classe) s'appelle l'encapsulation.

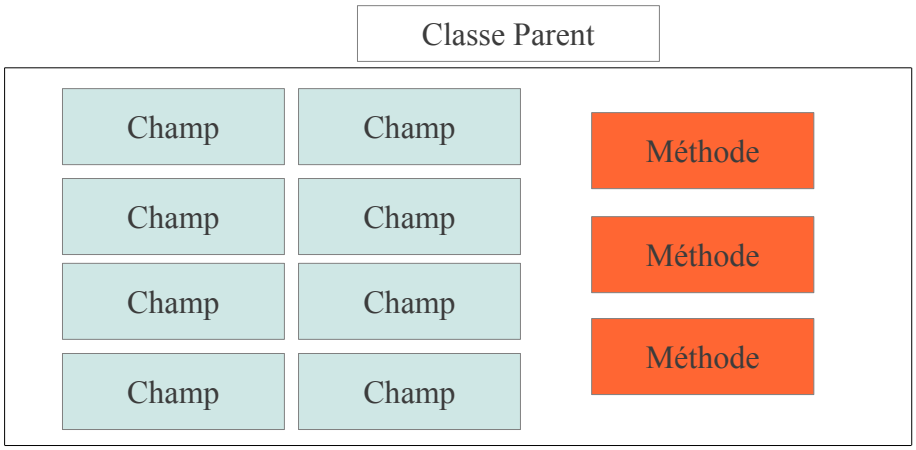
Une notion importante (après classe, encapsulation) est l'héritage. L'héritage est utile lorsqu'un objet peut être décliné en sous-groupe aillant des caractéristiques spécifiques. L'ensemble des attributs et méthodes communs à tous les sous groupes seront regroupés dans la classe mère (ou parent) et chaque sous groupe aura ses propres {attributs-méthodes} complémentaires définis dans une classe enfant. Lors de l'instanciation d'une classe enfant, on aura directement accès aux attributs/méthodes de cette classe enfant ainsi qu'aux attributs/méthodes de la classe parent dont il hérite automatiquement.

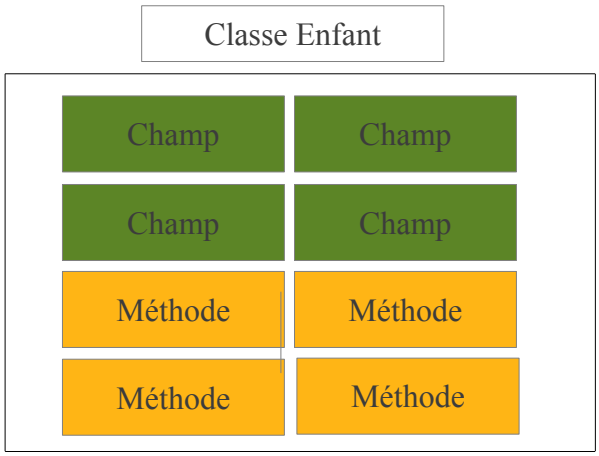
Note : Il existe des méthodes privées caractérisées pas des underscores (__nom__, cf. sec.8 p33)

Exemple : Parmi les articles on peut imaginer un traitement spécifique pour les articles périmés. Ces articles, en plus des attributs et méthodes propres à tous les articles, peuvent avoirs des informations supplémentaires telles que le lieu de recyclage où il faut les renvoyer, la perte

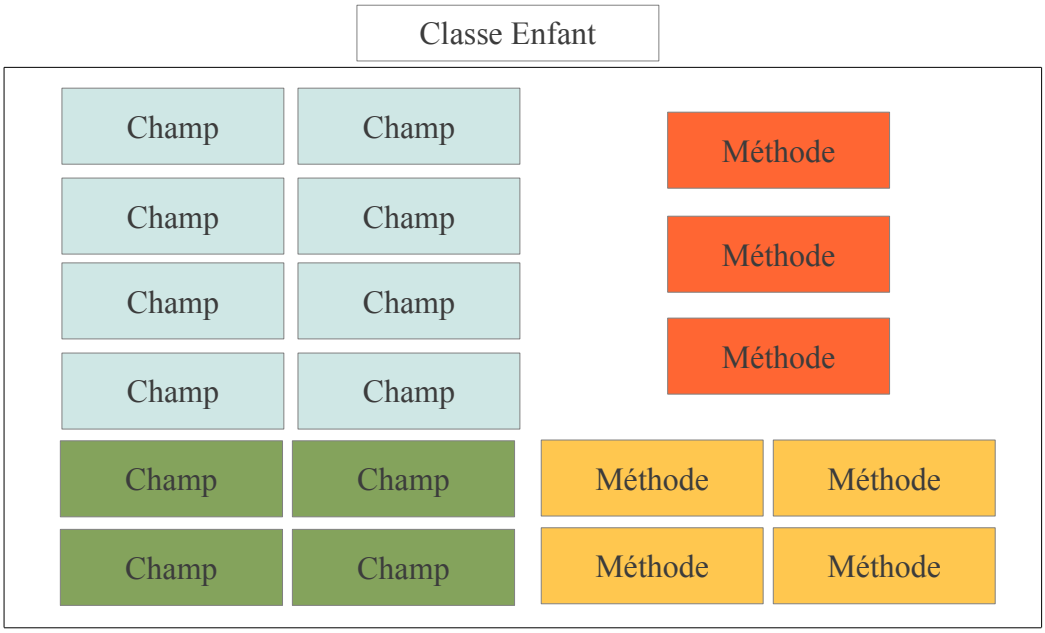
financière générée etc …

De façon générique l'héritage se présente de la manière suivante :





Ce qui équivaut à



Note : Dans une classe enfant il est possible de redéfinir ou modifier une méthode héritée de la classe parent pour l'adapter à l'utilisation spécifique pour la classe enfant. Cependant cela ne changera en rien les méthodes définies pour la classe parent à proprement parlé. Tout objet, instance de la classe parent, n'aura pas accès à la méthode modifiée.

3.2 - Définition de classe dans Python

Voici un exemple de l'implémentation d'une classe très basique. On a écrit dans un fichier nommé ClasseSimple.py la classe et on importe le module pour accéder à la classe.

```
class MaClasse:
    '" Cette classe est un exemple de définition avec un attribut et une méthode

    '"
    mon_pi = 3.14

    def message(self):
        print "Voici mon message"
```

```
In [1]: import ClasseSimple     # importation du module contenant la classe
In [2]: mc = ClasseSimple.MaClasse() # instanciation de la classe
In [3]: ? mc
Type:           instance
Base Class:     ClasseSimple.MaClasse
String Form:    <ClasseSimple.MaClasse instance at 0x1bc7830>
Namespace:      Interactive
File:           /home/fdelahaye/WORK/CoursCNAP/Python/Avance/ClasseSimple.py
Docstring:
    Cette classe est un exemple de définition avec un attribut et une méthode
In [5]: mc. [TAB]    #    On retrouve toutes les méthodes et attributs associés à la classe.
mc.__class__   mc.__doc__     mc.__module__  mc.message     mc.mon_pi
In [5]: mc.mon_pi
Out[5]: 3.1400000000000001
In [6]: mc.message()
Voici mon message
```

Note : On remarque qu'en plus de nos définitions (attributs et méthodes) il existe aussi quelques attributs déjà pré-définis. C'est les cas de *mc.__class__ mc.__doc__ mc.__module__*. Ces informations sont automatiquement crées lors du chargement du module. Le fait de déclarer une *classe* impose l'attachement de ces éléments de base à l'objet comme c'est le cas pour toute déclaration ou chargement d'objet. En effet dans Python tout objet est déjà défini de façon générique dans le noyau de Python et possède de facto des attributs et méthodes correspondant à sa nature. Lorsqu'on ecrit ' a=1' on instancie une classe de type 'integer' et un ensemble d'attributs/méthodes sont déjà prédéfinis. Il en est de même pour une classe. La classe que l'on déclare automatiquement hérite de la 'Super Classe' générique de python.

Rappel : Les attributs sont accessibles par l'appel de leur nom et les méthodes comme un appel de fonction (avec des parenthèses.

Note : Par convention toutes les méthodes déclarées dans une classe doivent avoir en premier argument *'self' (ou un autre mot, voir annexe B)*

3.3 - Porté des variables et objets - Scopes and Namespace

Attention, le problème de la portée des variables est cruciale de façon générale et peut être source de difficultés dans le contexte de la définition de classe. En annexe est reporté un paragraphe de la documentation du site python.org concernant les espaces et la portée des variables. A l'aide de quelques exemples nous allons essayer d'y voir plus clair.

On reprend la classe précédente :

```
class MaClasse:
    ''' Cette classe est un example de definition avec un atribut et une methode

    '''
    def mesVar(self) :
        pi = 3.14
        message = " Mon message"
        print pi, message

    def mon_pi(self):
        print pi
```

```
In [1]: import ClasseCercle
In [2]: mc = ClasseCercle.MaClasse()
In [3]: mc.mesVar()
3.14 Mon message
In [4]: mc.mon_pi()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/home/fdelahaye/WORK/CoursCNAP/Python/Avance/<ipython console> in <module>()
/home/fdelahaye/WORK/CoursCNAP/Python/Avance/ClasseCercle.pyc in mon_pi(self)
     24         print pi, message
     25
     26     def mon_pi(self):
---> 27                 print pi
     28
NameError: global name 'pi' is not defined
```

Dans le cas ci-dessus les variables définies dans les fonctions ont une portée se réduisant à la fonction uniquement. Si on veut utiliser une variable dans plusieurs méthodes de la classe il faut la définir hors des fonctions et la référencer dans les fonctions avec le préfixe *self.*

Dans l'exemple suivant dans la méthode mesVar on appelle self.pi et dans mon_pi on ne préfixe pas la variable.

```
class MaClasse:
    ''' Cette classe est un exemple de définition avec un attribut et une méthode
    '''
    pi = 3.14

    def mesVar(self) :
        print self.pi

    def mon_pi(self):
        print pi
```

```
In [1]: import ClasseSimple
In [2]: mc = ClasseSimple.MaClasse()
In [3]: mc.mesVar()
3.14
In [4]: mc.mon_pi()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/home/fdelahaye/WORK/CoursCNAP/Python/Avance/<ipython console> in <module>()
/home/fdelahaye/WORK/CoursCNAP/Python/Avance/ClasseCercle.py in mon_pi(self)
     23         print self.pi
     24
     25     def mon_pi(self):
---> 26                 print pi   # Il aurai fallu faire l'appel avec le préfixe self.
     27
NameError: global name 'pi' is not defined
```

3.4 - Instanciation et Initialisation

Après chargement du module contenant les classes, l'instanciation se fait par une affectation comme pour une fonction :' *manVarClass = mon_module.maClasse()* '. Il est possible de définir une méthode spéciale appelée '*__init__*' qui permet de faire ou définir un ensemble d'opérations qui s'exécuteront dès l'instanciation de la classe.

```
# fichier SimpleClasse.py
class MaClasse:
    ''' Cette classe est un exemple de définition avec un attribut et une méthode
    '''
    pi = 3.14

    def __init__(self) :
            import numpy as np
            print ' Mon pi etait', self.pi
            self.pi = np.pi
            print ' Mon pi est maintenant ', sef.pi

    def mon_message(self):
        message = '' Voici mon message ''
        print message
```

```
In [1]: import SimpleClasse
In [2]: mc = SimpleClasse.MaClasse()
Mon pi etait  3.14
Mon pi est maintenant  3.14159265359
In [3]: mc.mesVar()
3.14159265359
```

Note : Attention aux déclarations de variable de class et d'instance.

Si on déclare une variable de class *var1* et *var2* de la manière suivante :

```
# fichier class_simple.py
class MaClasse:
    ''' Cette classe est un exemple de définition avec un attribut et une méthode
    '''
    var1 = 3.14

    def __init__(self) :
        self.var2 = 2.
```

alors la variable *var1* est un variable de class et la variable *var2* est une variable d'instance. Ce qui signifie que le changement de sa valeur entrainera une modification de la valeur de var1 dans toute les instances de cette classe tandis que *var2* est indépendante dans chaque instance.

```
# dans ipython
In [1]: import class_simple as cl   # import du module contenant la classe
In [2]: c1=cl.MaClasse()   # premiere instance de la classe
In [3]: c2=cl.MaClasse()  # deuxieme instance de la classe
In [4]: print c1.var1, c1.var2, c2.var1, c2.var2
Out[4] : 1.0 2.0 1.0 2.0
In [5]: cl.MaClasse.var1 = 'Nouvelle Valeur'   # changement de la valeur de la var de class 'var1'
In [6]: print c1.var1, c1.var2, c2.var1, c2.var2
Out[6] : Nouvelle Valeur 2.0 Nouvelle Valeur 2.0
In [7]: cl.MaClasse.var2 = 'Et la var2?' # tentative sur la 'var2'
In [8]: print c1.var1, c1.var2, c2.var1, c2.var2
Out[8] : Nouvelle Valeur 2.0 Nouvelle Valeur 2.0 # cela n'affecte pas la valeur 'self.var2'
et ce même pour toute nouvelle instance.
In [9]: c3=cl.MaClasse()
In [10]: print c3.var1, c3.var2
Out[10] : Nouvelle Valeur 2.0
In [11]: c3.var2 = 'New var2'
In [12]: print c1.var2,  c3.var2
Out[12] : 2.0 New var2
```

Les variables de class de chaque instance pointent vers le même objet

```
# fichier class_simple.py
In [20]: id(cl.MaClasse.var1)==id(c1.var1)==id(c2.var1)==id(c3.var1)
Out[20]: True
In [21]: id(cl.MaClasse.var1),id(c1.var1),id(c2.var1),id(c3.var1)
Out[21]: (26408216, 26408216, 26408216, 26408216)
```

Note : Attention, si on change une valeur de classe dans une instance *c1.var1='new'*, alors cela revient à déclarer une nouvelle variable *var1* pour l'instance c1 de la classe et de fait celle-ci n'est plus reliée à la variable de classe *var1* initialement définie dans la classe. Donc tout changement de la variable de classe n'affectera plus *c1.var1*

Les variables d'instance (ayant le préfixe self.) représentent des objets différents. La valeur de ces variables n'est pas affectées dans la classe mais uniquement à l'instanciation. Donc il n'y a pas d'objet *class.var2* mais uniquement des objets *instance.var2*. Donc il n'est pas possible de modifier toutes les variables var2 de chaque instance d'un coup comme montré précédemment pour var1 (variable de classe).

La encore les variables *var2* des différentes instances pointent vers un même objet. Mais si on décide de changer la valeur dans l'une des instances alors, là encore, on crée un nouvel objet.

```
# fichier class_simple.py
In [4]: id(cl.MaClasse.var2)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/home/fdelahaye/Test_Spider/<ipython console> in <module>()
AttributeError: class MaClasse has no attribute 'var2'

In [5]: id(c1.var2),id(c2.var2),id(c3.var2)
Out[5]: (13653248, 13653248, 13653248)
In [7]: id(c1.var2)==id(c2.var2)==id(c3.var2)
Out[7]: True
In [8]: c1.var2 = 5
In [9]: id(c1.var2)==id(c2.var2)==id(c3.var2)
Out[9]: False
In [10]: id(c1.var2),id(c2.var2),id(c3.var2)
Out[10]: (11614040, 13653248, 13653248)
```

Il faut être rigoureux dans la déclaration et l'utilisation de variables pour ne pas perdre la valeur en cours de script et ne pas polluer inutilement l'environnement avec un nombre d'objets non utilisés.

3.5 - Héritage

```
class Article_Parent():
    ''' Ceci est la classe parent. Elle contient la définition d'un article.
    '''
    def __init__(self,nom):
                self.nom = nom
    def set_ref(self,ref):
        self.ref = ref

    def set_prixHT(self,prixHT):
            self.prixHT = prixHT

    def set_quantity(self,qty):
            self.qty = qty

    def add_stock(self,val):
            self.qty += val

    def remove_stock(self, val):
            self.qty -= val

    def prix_TTC(self):
            tva = 19.6
            prix_TTC = self.prixHT * (1. + tva/100.)
            return prix_TTC

    def prix_remise(self,taux_remise):
        prix_remise = self.prix_TTC() * (1. - taux_remise/100.)
        print r" Prix apres remise : %3.2f  : " %prix_remise

            #### Classe enfant #####
class ArtPerime_Enfant(Article_Parent):
    ''' Cette classe hérite de tous les attributs et méthodes de la classe
        parent et y ajoute ces propres attributs et méthodes.

    '''
    def set_DateLimite(self,jour,mois,annee):
            ''' La date limite doit être une liste de 3 éléments.
            '''
            import datetime as dt
        self.date_limite = dt.date(annee,mois,jour)
            today = dt.datetime.today()
            if self.date_limite < today.date():
            #print "L'article %s est perime. Nous le retirons du stock.",% self.nom
                self.remove_stock(self.qty)
```

Lors de la déclaration de la classe enfant on passe comme paramètre la classe parent. Ainsi cette classe enfant hérite automatiquement des attributs et méthodes de la classe parent.

```
import classeHeritage    # import du module contenant les deux classes Parent et Enfant

botte = classeHeritage.Article_Parent('botte')  # instanciation de la classe parent pour 1 article
botte.set_ref(1524)
botte.set_quantity(25)
botte.set_prixHT(100)
botte.prix_TTC()
botte.prix_remise(30.)

creme = classeHeritage.ArtPerime_Enfant('Creme Visage') # instanciation de la classe enfant
creme.set_quantity(60)
creme.set_prixHT(75.)
creme.qty
creme.set_DateLimite(12,1,2012)
creme.qty

gel = classeHeritage.ArtPerime_Enfant('Gel')   # instanciation de la classe enfant pour l'article gel
gel.set_quantity(60)
gel.set_prixHT(15.)
gel.qty
gel.set_DateLimite(12,1,2010)
```

Dans le script ci-dessus nous avons l'instanciation de la classe parent pour l'article *botte* et 2 instanciations de la classe enfant pour les articles *creme* et *gel*. L'instanciation de la classe enfant se fait de la même manière que la classe parent car elle a hérité de la méthode d'initialisation __init__.

Après avoir défini les dates de péremption pour les deux produits *creme* et *gel*, l'action de retirer du stock est automatiquement lancée comme on peut le voir ci-dessous dans l'exécution du script.

```
In [1]: import classeHeritage

In [3]: botte = classeHeritage.Article_Parent('botte')

In [4]: botte.set_ref(1524)

In [5]: botte.set_quantity(25)

In [6]: botte.set_prixHT(100)

In [7]: botte.prix_TTC()

Out[7]: 119.59999999999999

In [8]: botte.prix_remise(30.)

 Prix apres remise : 83.72  :

In [10]: creme = classeHeritage.ArtPerime_Enfant('Creme Visage')

In [11]: creme.set_quantity(60)

In [12]: creme.set_prixHT(75.)
```

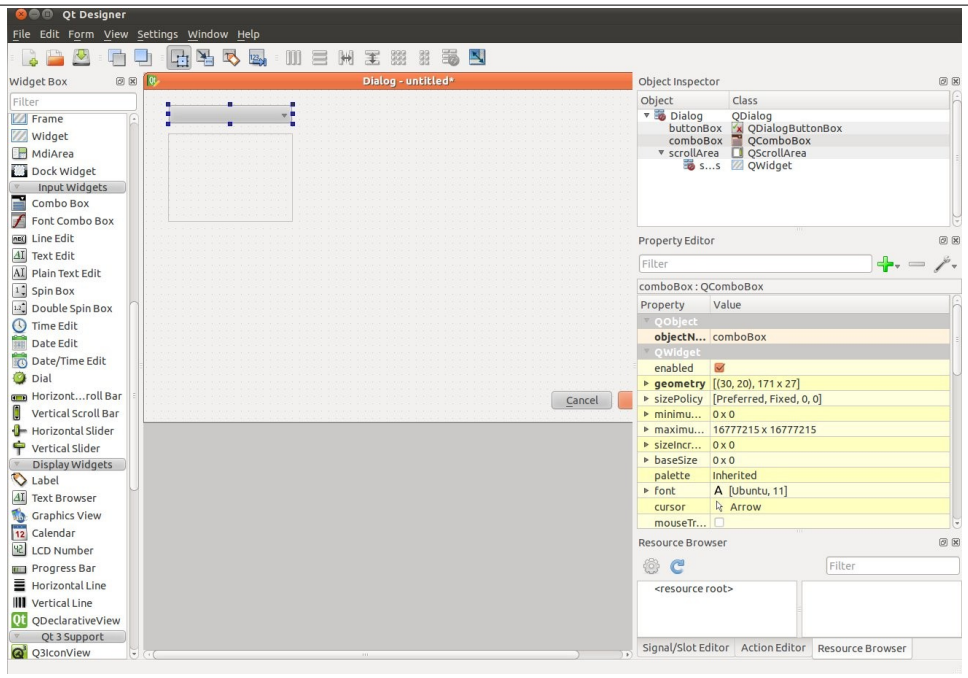
```
In [13]: creme.qty
Out[13]: 60
In [14]: creme.set_DateLimite(12,1,2012)
In [15]: creme.qty
Out[15]: 60
In [16]:
In [17]: gel = classeHeritage.ArtPerime_Enfant('Gel')
In [18]: gel.set_quantity(60)
In [19]: gel.set_prixHT(15.)
In [20]: gel.qty
Out[20]: 60
In [21]: gel.set_DateLimite(12,1,2010) # Date limite < Today alors qty = 0
In [22]: gel.qty
Out[22]: 0
```

4 - Python et les Widgets

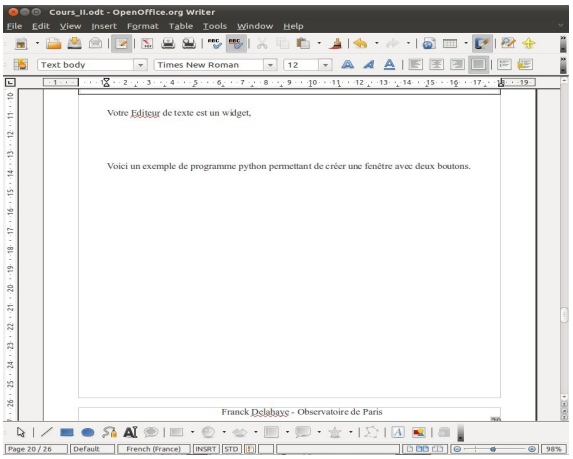
Afin de rendre plus simples, plus intuitives ou plus explicites certaines applications, on tend à s'éloigner de la ligne de commande qui nécessite la connaissance de tout un vocabulaire spécifique, une syntaxe parfois ésotérique pour basculer sur l'utilisation d'image, icône et autres boutons favorisant l'utilisation de la souris et une approche visuelle. Ce développement de l'utilisation d'une interface graphique entre l'utilisateur et la machine (interface Utilisateur-Machine ou Homme-Machine IHM, -Graphical User Interface -GUI- en anglais) se généralise pour les applications quelles qu'elles soient. L'ensemble des objets graphiques constituant une application appelé un widget.

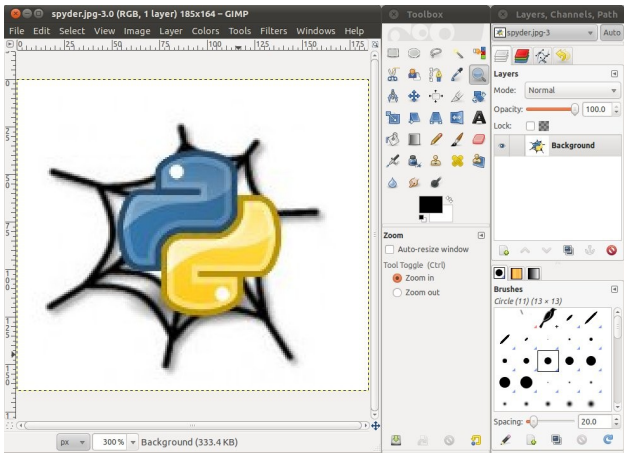
Ce passage de la ligne de commande à l'interface graphique se fait en plusieurs étapes. Une étape de 'design' permettant de définir la position relative des boutons et autres objets graphiques, leur apparence, ayant pour but d'optimiser l'ergonomie du widget. Ensuite il y a le développement de codes/scripts pour définir les actions de ces différents objet graphiques. Il existe plusieurs bibliothèques d'interfaces graphiques parmi lesquelles on peut nommer Tk, GTk, Qt. WxWidget. Celles ci permettent de créer des GUI à l'aide d'objets prédéfinis (boutons, zone de texte, menu ...). Il y existe des outils pour le design de GUI (QT designer, Wxglade …) permettant de construire des widgets de façon graphique, de visualiser et travailler sur la mise en place des différents éléments (boutons, menus, champs de saisie …) de façon simple et immédiate. Ces bibliothèques comprennent un ensemble d'objets (attributs et méthodes) qui peuvent être mis en œuvre pour la réalisation de la GUI. Ces bibliothèques sont accessible sous python via différents modules dérivés des bibliothèques graphiques initiales Tk ou Qt. Ainsi on peut nommer Tkinter, PyQt .

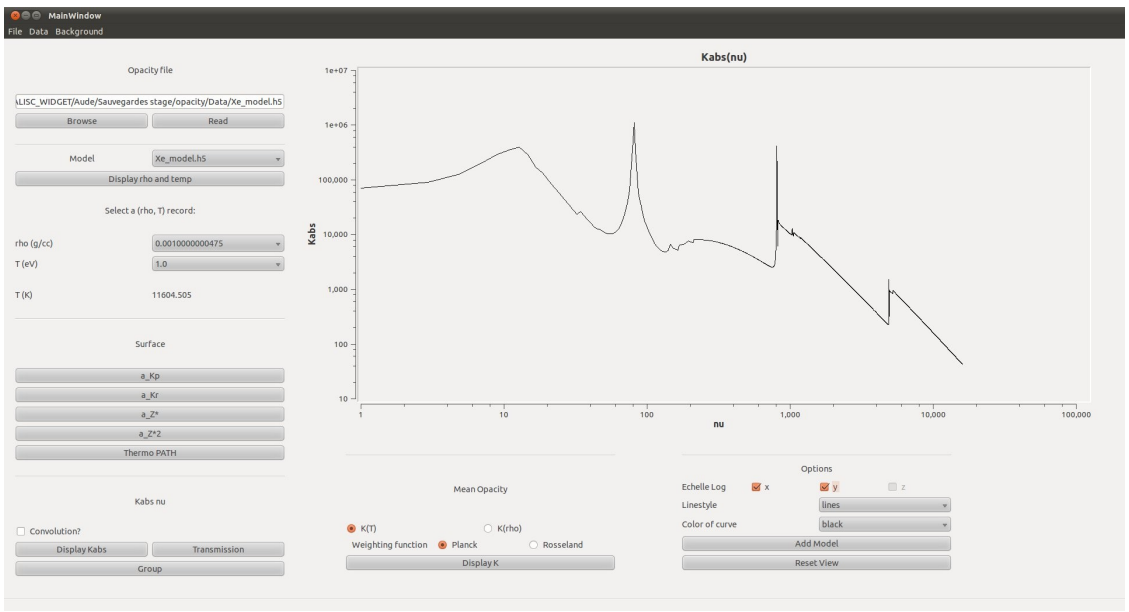
Voici un exemple de programme de création d'interfaces graphiques (Qt designer)



Éditeurs de texte, logiciels de traitement d'image ou applications spécifiques sont des widgets.







Pour la création d'un widget simple nous utiliserons directement la bibliothèques Tkinter dans le cadre de ce cours. Les liens vers les outils de design des widgets sont donnés à la fin de cette section.

Afin de se familiariser avec les différents composants voici un exemple simple de widget.

```
from Tkinter import *   # importation de tous les objets de Tkinter dans le namespace
class Application(Frame):     # Déclaration de la classe Application qui hérite des objet de Frame
    def say_hi(self):               # définition d'un fonction de l'application
        print "hi there, everyone!"

    def createWidgets(self):            # Définition de la fonction créant la partie graphique
        self.QUIT = Button(self)        # les boutons, leur nom et leur action.
        self.QUIT["text"] = "QUIT"
        self.QUIT["fg"]   = "red"
        self.QUIT["command"] =  self.quit
        self.QUIT.pack({"side": "left"})
        self.hi_there = Button(self)
        self.hi_there["text"] = "Hello",
        self.hi_there["command"] = self.say_hi
        self.hi_there.pack({"side": "left"})

    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.pack()
        self.createWidgets()

app = Application()
app.mainloop()
```

Bouton "QUIT"

Bouton "hi_there " (Attention le text sur le bouton est "Hello"

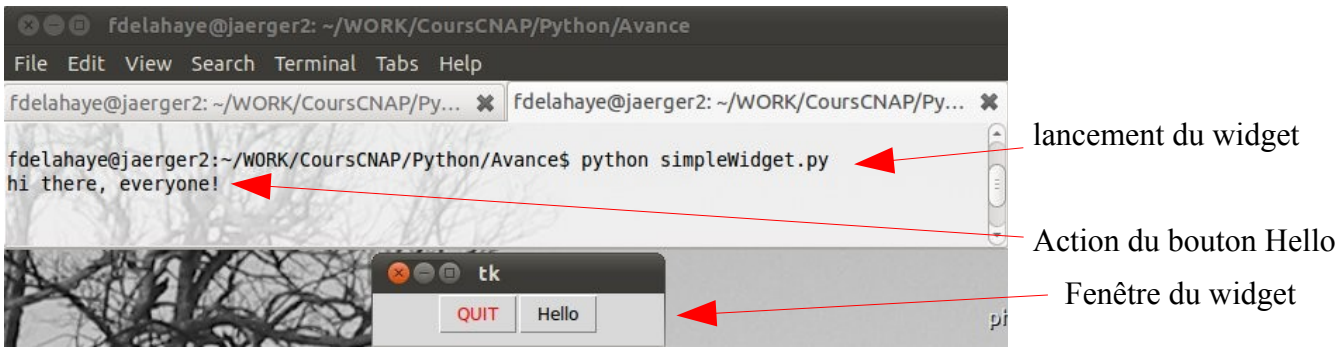
Ensemble d'actions lancées à l'initialisation

Instanciation

lancement de l'application

Sur le script précédent on remarque les différentes parties présentes dans la définition du widget. De façon générale chaque composant est comprend un ensemble d'attributs d'apparence (texte, couleurs, géométrie …) et d'action (positionnement, commande …).

python simpleWidget.py



Note : Dans le jargon des GUI, chaque entité ou objet (bouton, combobox, label …) est appelé widget. Ainsi on crée une application en assemblant plusieurs widgets de base.

Chacun des widgets de bases (wb pour la suite) sont des instances de sous classes d'une classe générique 'Widget' de Tkinter. Cette classe générique n'est pas instanciable à proprement parler mais regroupe les éléments communs à tous les wb. Lorsqu'on instancie un wb, on instancie une classe enfant de la classe 'Tkinter.Widget" .

On peut préciser les options du wb lors de l'instanciation ou a posteriori.

```
import Tkinter as tk
tk.Label(text="Texte du Label").pack()
tk.Button(text="text du Bouton").pack()          Instanciation avec option et mise en place
#--------------------------------- Ou ------------------------------------------------
import Tkinter as tk
lbl1 = tk.Label(text="Texte du Label")          Instanciation des wb
bt_quit = tk.Button()

bt_quit.config( text = "Quitter" )              Affectation des options
bt_quit['command'] = "quit"

lbl1.pack()                                     Mise en place dans la fenêtre principale
bt_quit.pack()
```

Sur l'affectation des options après instanciation d'un wb on peut le faire de 2 manières différentes (illustrées ci-dessus). On peut obtenir les différentes options accessibles par l'aide en ligne sur l'instance du wb.

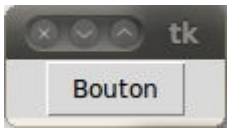
```
In [11]: ? bt_quit
String Form:   .21959456
Namespace:     Interactive
File:          /usr/lib/python2.6/lib-tk/Tkinter.py
Docstring:
   Button widget.
Constructor Docstring:
   Construct a button widget with the parent MASTER.
   STANDARD OPTIONS
      activebackground, activeforeground, anchor,
      background, bitmap, borderwidth, cursor, disabledforeground, font, foreground
      highlightbackground, highlightcolor, highlightthickness, image, justify,
      padx, pady, relief, repeatdelay, repeatinterval, takefocus, text, textvariable,
      underline, wraplength
   WIDGET-SPECIFIC OPTIONS
      command, compound, default, height,
      overrelief, state, width
```

Dans le cas présent, la mise en place est déléguée à un composant invisible appelé 'packer' qui gère automatiquement la disposition relative des wbs. (pack()). Cependant il est possible de passer des paramètres pour une disposition personnalisée.

4.1 - Widget de base - wb

```
Import Tkinter as tk
w1 = tk.Tk()
wb_label = tk.Label(w1,text="Label").pack()
```


```
w2 = tk.Tk()
wb_bouton = tk.Button(w2,text = "Bouton").pack()
```

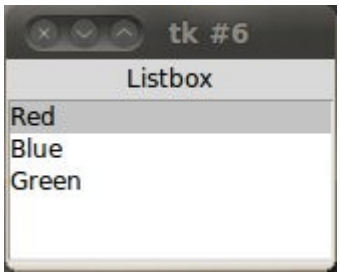



```
w3= tk.Tk()
lbl_saisie = tk.Label(w3, text="Champ de saisie").pack(side=tk.LEFT)
wb_champSaisie = tk.Entry(w3)
wb_champSaisie.pack()
```


```
w4=tk.Tk()
lbl_rb = tk.Label(w4, text="Bouton radio").pack(side=tk.LEFT)
wb_RadioButton = tk.Radiobutton(w4)
wb_RadioButton.pack()
```


```
w5=tk.Tk()
lbl_rb = tk.Label(w5, text="Spinbox").pack(side=tk.LEFT)
wb_Spinbox = tk.Spinbox(w5,values=("Square","Triangle","Circle"))
wb_Spinbox
```


```
w6=tk.Tk()
lbl_rb = tk.Label(w6, text="Listbox")
wb_List = tk.Listbox(w6)
wb_List.insert(tk.END,"Red")
wb_List.insert(tk.END,"Blue")
wb_List.insert(tk.END,"Green")
wb_List.pack()
```




```
w7=Tk()
w7.geometry("200x100")
pane=tk.PanedWindow(orient=HORIZONTAL) ;# ou VERTICAL
pane.pack(expand="yes",fill="both")
left=tk.Label(pane,text="Côté gauche",bg="yellow")
right=tk.Label(pane,text="Côté droite",bg="white")
pane.add(left)
pane.add(right)
w7.mainloop()
```


```
w8=tk.Frame()  # c'est juste un cadre (visible ou invisible) qui joue le rôle de conteneur et permet
                de délimiter les différentes zone de la fenêtre d'application.
```

4.2 - Le placement des wbs dans la fenêtre

Pour l'organisation des wb sur la fenetre principale, il existe plusieurs méthodes. Celles ci sont associées automatiquement au wb. Les deux principaux gestionnaires de positionnement sont le 'packer' et le 'gridder'. On y accède par les méthodes respectives .pack() et .grid().

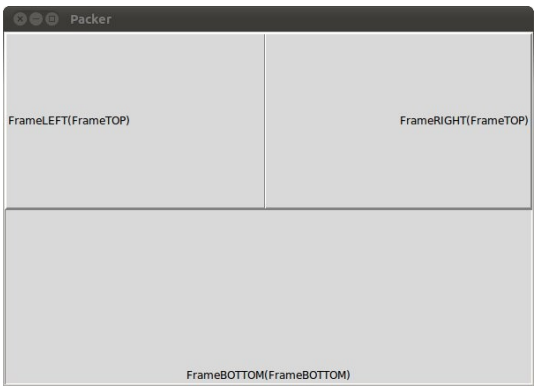
4.2.1 - Le Packer :

Syntaxe – *wb.pack(**option)*
rappel: **option signifie le passage de paramètres optionnels sous forme *wb.pack(param1 = val1, param2 = val2 ..)*

Les paramètres possibles :

side = LEFT ou RIGHT ou TOP ou BOTTOM
Il détermine le bord contre lequel se placera le wb dans son wb_parent.
expand = 0 ou 1. Si 1 wb s 'étend pour remplir l'espace de son parent.
fill = NONE, X, Y, BOTH. Remplit l'espace supplémentaire alloué par le packer en plus des dimensions minimales du wb dans la(les) direction(s) indiquée(s) par la valeur du paramètre.

```
# -*- coding: utf-8 -*-
import Tkinter as tk
class MesPositions:

    def __init__(self,masterW):
        self.placement(masterW)

    def placement(self,masterW):
        fr_principal = tk.Frame(masterW,borderwidth=1)
        fr_Upper = tk.Frame(fr_principal,borderwidth=1,relief = "raised")
        fr_Lower = tk.Frame(fr_principal,borderwidth=1,relief = "sunken")
        fr_Uleft = tk.Frame(fr_Upper,borderwidth=1,relief = "raised")
        fr_Uright = tk.Frame(fr_Upper,borderwidth=1,relief = "raised")

        lbl_UL=tk.Label(fr_Uleft,text="FrameLEFT(FrameTOP)").pack(side=tk.LEFT)
        lbl_UR=tk.Label(fr_Uright,text="FrameRIGHT(FrameTOP)").pack(side=tk.RIGHT)
        lbl_BB=tk.Label(fr_Lower,text="FrameBOTTOM(FrameBOTTOM)").pack(side=tk.BOTTOM)

        fr_principal.pack(side=tk.TOP, expand=1,fill=tk.BOTH)
        fr_Upper.pack(side = tk.TOP, expand=1,fill=tk.BOTH)
        fr_Lower.pack(side = tk.BOTTOM, expand=1,fill=tk.BOTH)
        fr_Uleft.pack(side = tk.LEFT, expand=1,fill=tk.BOTH)
        fr_Uright.pack(side = tk.RIGHT, expand=1,fill=tk.BOTH)

def main():
    masterW=tk.Tk()
    masterW.title("Packer")
    masterW.geometry("600x400")
    mwidget = MesPositions(masterW)
    masterW.mainloop()

main()
```



Il est assez pratique de faire la mise en place des wbs avec un ensemble de *frame* parents qui jouent le rôle de conteneur de wb et délimitent ainsi les différents espaces de la fenêtre.

4.2.2 - Le gridder

Le gridder utilise une grille sur laquelle on place les wbs par des coordonnées de lignes et colonnes et pour lesquelles on explicite la place occupée.

Syntaxe – *wb.grid(**option)*

Les paramètres possibles :

column : Colonne dans laquelle est placé le wb

columspan : Taille en nombre de colonnes

row : Ligne dans laquelle est placé le wb

rowspan : Taille en nombre de lignes

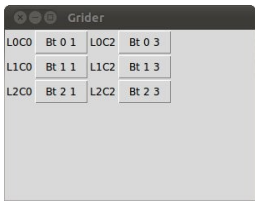
ipadx,ipady : Nombre de pixels (en x et y respectivement) pour étendre le wb. La cellule s'adapte à cette nouvelle taille.

padx, pady : Nombre de pixels (en x et y respectivement) pour étendre la cellule du wb. Ceci revient à augmenter l'espace entre plusieurs wbs

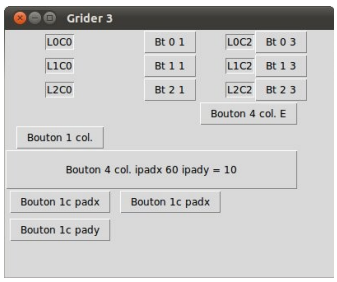
sticky = ' ', N,E,S,W,NE,SW... Indique où coller le wb à l'intérieur de la cellule. Les points cardinaux étant associés aux bords ou coins de la cellule. ' ' indique centré dans la cellule.

Si aucune option n'est indiquée, alors la prochaine cellule disponible qui est utilisée.

Attention, un wb a une dimension minimale, celle ci déterminera la taille de la cellule. Si différents wbs ont des tailles différentes, alors la taille de la colonne ou ligne prendra la taille de la taille du plus grand wb présent dans la même colonne ou ligne. Voire les exemples ci dessous.



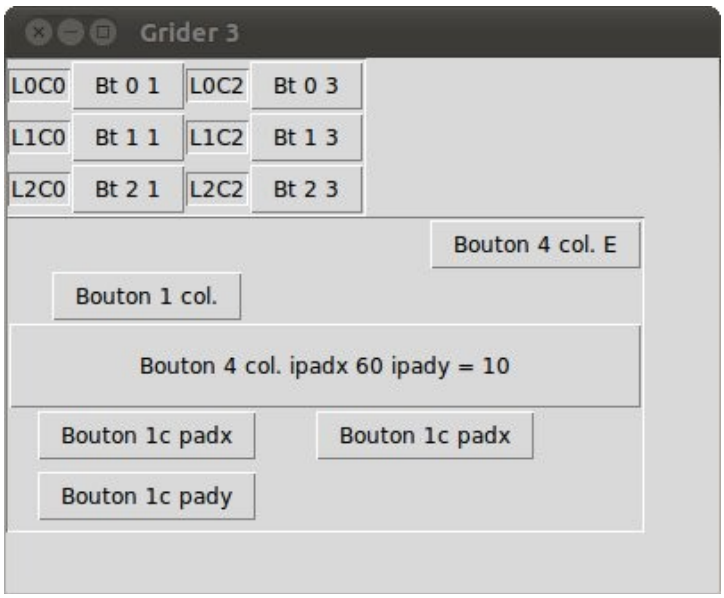

```
class MesPositions2:
   def __init__(self,masterW):

      self.placement(masterW)
   def placement(self,masterW):
      for l in range(3):
         for c in range(0,4,2):
            tk.Label(masterW,text='L%sC%s' %(l,c),relief='sunken').grid(row=l,column=c)
         for c in range(1,4,2):
            tk.Button(masterW,text='Bt %s %s' %(l,c)).grid(row=l,column=c)
            #tk.Button(masterW,text='Bt %s %s' %(l,c)).grid(row=l,column=c,columnspan = cspan,ipady = 20)
```




```
class MesPositions3:
   def __init__(self,masterW):

      self.placement(masterW)
   def placement(self,masterW):
      for l in range(3):
         for c in range(0,4,2):
            tk.Label(masterW,text='L%sC%s' %(l,c),relief='sunken').grid(row=l,column=c)
         for c in range(1,4,2):
            cspan = 1
            if c ==3:cspan = 2
            tk.Button(masterW,text='Bt %s %s' %(l,c)).grid(row=l,column=c,columnspan = cspan)
      tk.Button(masterW,text='Bouton 4 col. E').grid(columnspan = 4,sticky=tk.E)
      tk.Button(masterW,text='Bouton 1 col.').grid()
      tk.Button(masterW,text='Bouton 4 col. ipadx 60 ipady = 10').grid(columnspan = 4,ipadx = 60,ipady = 10)
      tk.Button(masterW,text='Bouton 1c padx').grid(row=7,padx = 5)
      tk.Button(masterW,text='Bouton 1c padx').grid(row=7,column = 1, padx=5)
      tk.Button(masterW,text='Bouton 1c pady').grid(row=8,column = 0, pady=5)
```

Bien sûr on peut combiner le gridder avec un ensemble de cadres, ainsi les redimensionnements de cellule imposés par les wbs ou modifications par le gridder n'affectent que la grille associée à un cadre spécifique.

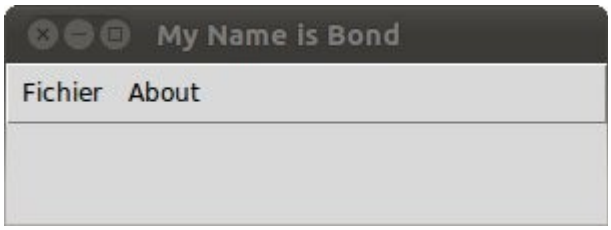
```
class MesPositions4:

    def __init__(self,masterW):
        self.placement(masterW)

    def placement(self,masterW):
        frame1 = tk.Frame(masterW,borderwidth=1,relief='sunken')
        frame2 = tk.Frame(masterW,borderwidth=1,relief = 'sunken')
        for l in range(3):
            for c in range(0,4,2):
                tk.Label(frame1,text='L%sC%s' %(l,c),relief='sunken').grid(row=l,column=c)
            for c in range(1,4,2):
                cspan = 1
                if c ==3:cspan = 2
                tk.Button(frame1,text='Bt %s %s' %(l,c)).grid(row=l,column=c,columnspan = cspan)
        tk.Button(frame2,text='Bouton 4 col. E').grid(columnspan = 4,sticky=tk.E)
        tk.Button(frame2,text='Bouton 1 col.').grid()
        tk.Button(frame2,text='Bouton 4 col. ipadx 60 ipady = 10').grid(columnspan = 4,ipadx = 60,ipady = 10)
        tk.Button(frame2,text='Bouton 1c padx').grid(row=7,padx = 5)
        tk.Button(frame2,text='Bouton 1c padx').grid(row=7,column = 1, padx=5)
        tk.Button(frame2,text='Bouton 1c pady').grid(row=8,column = 0, pady=5)
        frame1.grid(sticky=tk.W)
        frame2.grid()
```



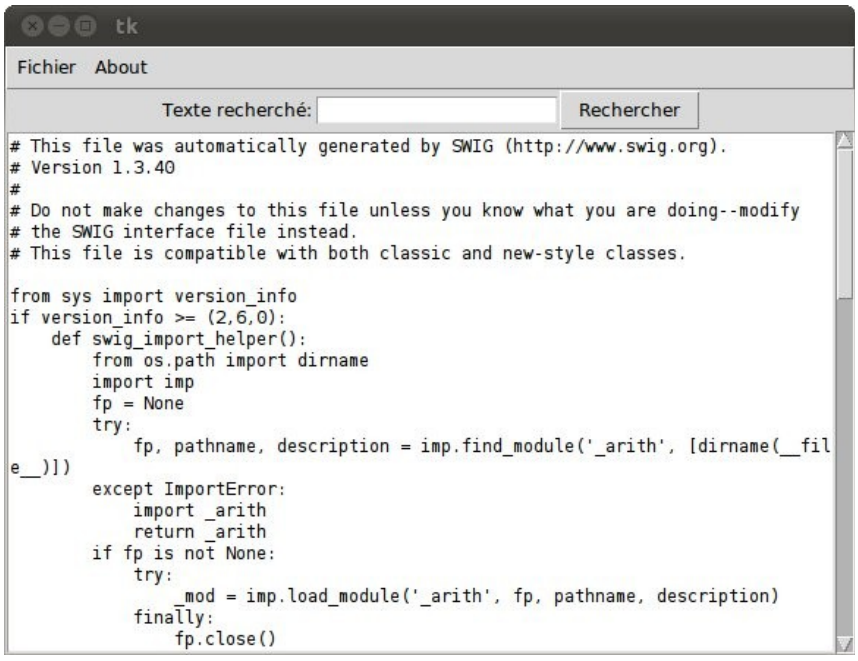
4.3 - Les menus

Dans les exemples suivants nous implémentons des menus et sous menus avec des commandes associées.

```
w8 = tk.Tk()
w8.geometry("300x50")

# Définition d'une action
def chtitre(choix):
    w8.title(choix)

# Création de la barre de menu:
menuP = tk.Menu(w8) # instanciation d'un objet menu associer au wb w8 » qui est la fenêtre principale.

# Création du menu fichier:
mfichier = tk.Menu(menuP, tearoff=0) # instanciation d'un objet menu 'mfichier' associer au menuP
menuP.add_cascade(label="Fichier",menu=mfichier) # ajout de l'objet mfichier dans le menuP
mfichier.add_command(label="Quit", command=w8.destroy) # ajout d'une commande dans le menu mfichier

# Création du menu About
mAbout = tk.Menu(menuP, tearoff=0) # instanciation d'un objet menu 'mAbout' associer au menuP
menuP.add_cascade(label="About",menu=mAbout) # ajout de l'objet mfichier dans le menuP
# Création du sous menu Auteur dans le menu About
smAuteur = tk.Menu(mAbout, tearoff=0)
mAbout.add_cascade(label="Auteur",menu=smAuteur)
# Création des sous sous menu Nom et Prénom auquelle on affecte une fonction.
smAuteur.add_command(label="Nom", command=lambda  : chtitre('My Name is Bond'))
smAuteur.add_command(label="Prenom", command=lambda : chtitre('James Bond'))

# afficher le menu
w8.config(menu=menuP)
w8.mainloop()
```



Voici un exemple de widget plus compliqué combinant plusieurs éléments. Dans le code correspondant (présenté dans les pages suivantes) les actions du champ de saisie et du bouton 'rechercher' n'ont pas été codés. Cela est réservé pour le TP.



```
import Tkinter as tk
# -*- coding: utf-8 -*-
import Tkinter as tk
from tkFileDialog import askopenfilename

class Mon_Application:
  ####        Partie de design            ####
  # Fenetre Principale
  def __init__(self,masterW):
    ''' Initialisation: crée tous les éléments du widget à l'instanciation. '''
    mainW = tk.Frame(masterW)
    self.MakeMenubar(masterW)
    self.MakeUpper(mainW)
    self.MakeLower(mainW)
    mainW.pack()
    return

  # Menu
  def MakeMenubar(self,mainW):
    ''' Barre des menu de la fenetre principale. '''
    m_menuP = tk.Menu(mainW)
    m_fichier = tk.Menu(m_menuP, tearoff = 0)
    m_menuP.add_cascade(label="Fichier",menu = m_fichier)
    m_fichier.add_command(label="Ouvrir", command = lambda: self.cmd_ouvrir())
    m_fichier.add_command(label="Quitter", command = mainW.destroy)
    m_About = tk.Menu(m_menuP, tearoff = 0)
    m_menuP.add_cascade(label="About",menu = m_About)
    m_About.add_command(label="Version", command=lambda: self.cmd_version())
    mainW.config(menu = m_menuP)

  def MakeUpper(self,mainW):
    ''' Partie Supérieur de la fenetre principale '''
    cadreS = tk.Frame(mainW)
    self.lbl_saisie = tk.Label(cadreS)
    self.lbl_saisie['text'] = "Texte recherché:"
    self.ent_saisie = tk.Entry(cadreS)
    self.bt_recherche = tk.Button(cadreS)
    self.bt_recherche['text'] = "Rechercher"
    # Positionement de tous les elements de la partie suupérieure
    cadreS.pack(side=tk.TOP)
    self.lbl_saisie.pack(side=tk.LEFT)
    self.ent_saisie.pack(side=tk.LEFT)
    self.bt_recherche.pack(side=tk.RIGHT)
    cadreS.pack(side=tk.TOP)

  def MakeLower(self,mainW):
    ''' Partie inferieure de la fenêtre principale. '''
    cadreI = tk.Frame(mainW)
    self.txt_texte = tk.Text(cadreI,background='white')
    self.txt_texte.insert('1.0',''' Cecie est la zone de texte''')
    scroll = tk.Scrollbar(cadreI)
    self.txt_texte.configure(yscrollcommand = scroll.set)
    self.txt_texte.pack(side = tk.LEFT)
    scroll.pack(side = tk.RIGHT,fill = tk.Y)
    cadreI.pack(side = tk.BOTTOM)
    return
```

```
    ####      Partie comportements et actions
    def cmd_ouvrir(self):
        ''' Commande d'ouverture d'un fichier et affichage du contenu dans la zone texte.

        '''
        filename = askopenfilename(filetypes=[("allfiles","*"),("pythonfiles","*.py")])
        myfile = open(filename,'r')
        txt = myfile.read()
        self.txt_texte.insert('1.0',' ')
        self.txt_texte.insert('1.0',txt)
        return filename

    def cmd_version(self):
        ''' Ouvre une fenetre et affiche la vesion.

        '''
        msg_mas = tk.Tk()
        msg_lbl = tk.Label(msg_mas)
        msg_lbl['text'] =  "Ceci est la version 0.0.1"
        msg_lbl.pack()
        msg_mas.mainloop()

def main():
    ''' Ceci est l'ensemble des commandes de lancement de l'application.

    '''
    masterW=tk.Tk()
    mwidget = Mon_Application(masterW)
    masterW.title = "Mon Application"
    masterW.geometry ="600x400"
    masterW.mainloop()

# execution de l'application
main()
```

PyQt n'est pas abordé ici mais l'esprit reste le même. Il existe là encore de nombreuses références dont voici un extrait très restreint. Celles ci m'ont permis préparer ce cours.

"Python en concentré – Manuel de référence" A. Martelli, ed. O'Reilly (Présent dans la bibliothèque de l'Observatoire de Paris)

http://gnuprog.info/prog/python/pwidget.php

http://docs.python.org/library/tkinter.html

5 - Conclusion

Ce cours n'est pas une synthèse. L'objectif est juste de proposer un support initial pour avoir une vision de la programmation en Python, sa simplicité et sa puissance. L'importance de la documentation existante en fait un outil en très accessible quelque soit son niveau de programmation.

6 - Annexe A : Scopes et Namespace

Le fonctionnement de python et plus particulièrement les définitions de champs d'application ou de porté des variables, méthodes ou classes est essentielle pour le bon fonctionnement des scripts.

La documentation originale sur ce sujet sur le site python.org est à cette adresse : http://docs.python.org/tutorial/classes.html.

J'ai reproduit ci-dessous le paragraphe concerné.

Let's begin with some definitions.

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`*, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function* `maximize` *without confusion — users of the modules must prefix it with the module name.*

By the way, I use the word attribute for any name following a dot — for example, in the expression `z.real`*,* `real` *is an attribute of the object* `z`*. Strictly speaking, references to names in modules are attribute references: in the expression* `modname.funcname`*,* `modname` *is a module object and* `funcname` *is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace! [1]*

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`*. Writable attributes may also be deleted with the* `del` *statement. For example,* `del modname.the_answer` *will remove the attribute* `the_answer` *from the object named by* `modname`*.*

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`*, so they have their own global namespace. (The built-in names actually also live in a module; this is called* `__builtin__`*.)*

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A scope is a textual region of a Python program where a namespace is directly accessible. "Directly accessible" here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- *the innermost scope, which is searched first, contains the local names*
- *the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names*
- *the next-to-last scope contains the current module's global names*
- *the outermost scope (searched last) is the namespace containing built-in names*

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at "compile" time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` *statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement* `del x` *removes the binding of* `x` *from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular,* `import` *statements and function definitions bind the module or function name in the local scope. (The* `global` *statement can be used to indicate that particular variables live in the global scope.)*

7 - Annexe B : Quelques notes sur la syntaxe de classe.

7.1 - Variable dans les classes.

Note on Instance of a variable and Class variable.

A.x is a class variable. B's self.x is a instance variable.

i.e. A's x is shared between instances.

It would be easier to demonstrate the difference with something that can be modified like a list:

```
#!/usr/bin/env python
class A:
    x = []
    def add(self):
        self.x.append(1)
class B:
    def __init__(self):
        self.x = []
    def add(self):
        self.x.append(1)
x = A()
y = A()
x.add()
y.add()
print "A's x:",x.x
x = B()
y = B()
x.add()
y.add()
print "B's x:",x.x
```

Output

```
A's x: [1, 1]
B's x: [1]
```

A.x is a class variable, and will be shared across all instances of A, unless specifically overridden within an instance. B.x is an instance variable, and each instance of B has its own version of it.

I hope the following Python example can clarify:

```
>>> class Foo():
...     i = 3
...     def bar(self):
...             print 'Foo.i is', Foo.i
...             print 'self.i is', self.i
...
>>> f = Foo() # Create an instance of the Foo class
>>> f.bar()
Foo.i is 3
self.i is 3
>>> Foo.i = 5 # Change the global value of Foo.i over all instances
>>> f.bar()
Foo.i is 5
self.i is 5
>>> f.i = 3 # Override this instance's definition of i
>>> f.bar()
Foo.i is 5
self.i is 3
```

7.2 - Self

Just as a side note: `self` is actually just a randomly chosen word, that everyone uses, but you could also use `this`, `foo`, or `myself` or anything else you want, it's just the first parameter of every non static method for a class. This means that the word `self` is not a language construct but just a name:

```
>>> class A:
...     def __init__(s):
...         s.bla = 2
...
>>>
>>> a = A()
>>> a.bla
2
```

8 - Note On the Underscore :

Difference between _, __ and __xx__ in Python

16 Set 2010

When learning Python many people don't really understand why so much underlines in the beginning of the methods, sometimes even in the end like __this__! I've already had to explain it so many times, it's time to document it.

One underline in the beginning

Python doesn't have real private methods, so one underline in the beginning of a method or attribute means you shouldn't access this method, because it's not part of the API. It's very common when using properties:

```
class BaseForm(StrAndUnicode):
    ...
    def _get_errors(self):
        "Returns an ErrorDict for the data provided for the form"
        if self._errors is None:
            self.full_clean()
        return self._errors
    errors = property(_get_errors)
```

This snippet was taken from django source code (django/forms/forms.py). This means errors is a property, and it's part of the API, but the method this property calls, _get_errors, is "private", so you shouldn't access it.

Two underlines in the beginning

This one causes a lot of confusion. It should not be used to mark a method as private, the goal here is to avoid your method to be overriden by a subclass. Let's see an example:

```
class A(object):
  def __method(self):
     print "I'm a method in A"
  def method(self):
     self.__method()
>>a = A()
>>a.method()
```

The output here is

```
$ python example.py
I'm a method in A
```

Fine, as we expected. Now let's subclass A and customize __method

```
class B(A):
  def __method(self):
     print "I'm a method in B"
b = B()
b.method()
```

and now the output is...

```
$ python example.py
I'm a method in A
```

as you can see, A.method() didn't call B.__method() as we could expecte. Actually this is the correct behavior for __. So when you create a method starting with __ you're saying that you don't want anybody to override it, it will be accessible just from inside the own class.

How python does it? Simple, it just renames the method. Take a look:

```
a = A()
a._A__method()  # never use this!! please!
```

```
$ python example.py
I'm a method in A
```

If you try to access a.__method() it won't work either, as I said, __method is just accessible inside the class itself.

Two underlines in the beginning and in the end

When you see a method like __this__, the rule is simple: don't call it. Why? Because it means it's a method python calls, not you. Take a look:

```
>>> name = "igor"
>>> name.__len__()
4
>>> len(name)
4
>>> number = 10
>>> number.__add__(20)
30
>>> number + 20
30
```

There is always an operator or native function that calls these magic methods. The idea here is to give you the ability to override operators in your own classes. Sometimes it's just a hook python calls in specific situations. __init__(), for example, is called when the object is created so you can initialize it. __new__() is called to build the instance, and so on...

Here's an example:

```
class CrazyNumber(object):
  def __init__(self, n):
     self.n = n
  def __add__(self, other):
     return self.n - other
  def __sub__(self, other):
     return self.n + other
  def __str__(self):
      return str(self.n)
num = CrazyNumber(10)
print num          # 10
print num + 5      # 5
print num - 20     # 30
```

Another example:

```
class Room(object):
    def __init__(self):
        self.people = []
    def add(self, person):
        self.people.append(person)
    def __len__(self):
        return len(self.people)
room = Room()
room.add("Igor")
print len(room)     # 1
```

The documentation covers all these special methods.

Conclusion

Use _one_underline to mark you methods as not part of the API. Use __two_underlines__ when you're creating objects to look like native python objects or you wan't to customize behavior in specific situations. And don't use __just_to_underlines, unless you really know what you're doing!

Aknowledgement : Merci à Christian et Nicolas pour leur relecture attentive permettant de réduire sérieusement le nombre de fautes et d'éclaircir certains points.