Memento

Ce memento est téléchargeable et imprimable sur le site.

25.1 Généralités

25.1.1 Help

Dans le module **built-in** (automatiquement chargé au démarrage de Pyhton, voir plus bas), il existe un système d'aide assez performant.

Tu es dans la console : tape

```
>>>help()
```

et voilà ce qui sort… traduit en français…

```
Bienvenue dans Python 2.7 ! Voici l'utilitaire d'aide en ligne.
Si tu utilises Python pour la première fois, tu devrais consulter le
tutoriel à l'adresse http://docs.python.org/tutorial/.
```

```
Entre le nom de n'importe quel module, mot clef, sujet pour obtenir de
l'aide sur l'écriture de programmes en Python ainsi que sur les
modules. Pour quitter cet utilitaire et retourner à l'interpréteur,
tape simplement « quit ».
Pour obtenir une liste des modules disponibles, mots clefs ou sujets,
tape "modules", "keywords", ou "topics". Chaque module affiche un
petit résumé de son usage ; pour lister les modules dont le résumé
contient un mot donné comme « spam », tape "modules spam".
```

Après essai il s'avère que ça ne marche pas dans Pyscripter, par contre si tu tapes

```
>>> help('modules')
```

ça marche très bien et tu obtiens la liste de tous les modules présents. Pour les infos sur un module, tape

```
>>> import nomModule
>>> help(nomModule) ou help(nomModule.nomFonction)
```

Tu peux également lancer un script avec ces instructions.

Comme quasiment tout est en anglais le résultat n'est pas toujours facile à comprendre, mais ça peut aider pour rappeler la syntaxe d'une fonction par exemple.

On peut taper `dir(nomModule)` à la place de `help(nomModule)` : ça donnera juste la liste des fonctions disponibles.

25.1.2 Documentation et commentaires

Rappelons ce qui a été dit : mettre des commentaires de manière systématique pour expliquer ce qu'on fait ; il est inutile de commenter toutes les opérations mais de dire plutôt pourquoi on le fait. Rappelle-toi : les commentaires doivent t'aider, ou quelqu'un à qui tu passeras ton script, à comprendre et te rappeler ce qui s'y passe… surtout après qu'un certain laps de temps se soit écoulé !

Commentaire sur une seule ligne :

```
# bla bla bla
```

ou encore

```
print 'Je suis beau'  # on veut des preuves
```

On peut mettre plus d'un #, ça fera changer de couleur.

Sur plusieurs lignes : triple guillemets `" " "` ou triple apostrophe `' ' '` encadrent plusieurs lignes (pas les mêmes couleurs suivant les éditeurs) :

```
" " " Ceci est un commentaire : situé juste après
une fonction, il servira de docstring, très utile" " "
```

25.2 Éléments de syntaxe

Les mots réservés de Python et leur traduction (entre parenthèses le mot complet si abrégé) :

and	et	or	ou	not	pas / non
for	pour	while	tant que	yield	valeur
del (delete)	efface	print	imprime (affiche)	from	depuis
def (define)	définit	return	retourne	global	global
pass	passer	break	arrêter	continue	continuer
raise	remonte	try	essaie	except	exception
None	Rien	True	Vrai	False	Faux
in	dans	import	importe	as	comme
if	si	elif (else if)	sinon si	else	sinon
exec (execute)	exécuter	eval (evaluate)	évaluer		

25.2.1 Données et opérateurs

Typage des données (exemples en commentaires)

chaîne de caractères : **str**() #st="abcdef" ou 'abcdef' / on utilise aussi **unicode**()	
entier : **int**() ou **long**() #n=125L, pas de max.	booléen : **bool**() #True ou False
pseudo-réel : **float**() #12.5, max=1e302	complexe : **complex**() #5+2j ; i pas utilisé.
indexation : *liste*[i] #L[i]	segmentation : *liste*[i:j] #L[i:j]
Attention : a = b n'est pas un operateur math. mais une **assignation** de variable.	

Opérations

Mathématiques	**Evaluations**
+ - * / # division flottante	a == 10 # égalité mathématique
// #division entière (version <=2.7)	a != b # différent de
% #modulo	a > b # plus grand que (strict)
** # puissance	a < b # plus petit que (strict)
	a >= b # plus grand ou égal
Logique : not, and, or, in	a <= b # plus petit ou égal

Les opérateurs de comparaison sont utilisables pour des tris alphabétiques. **in** est valable sur les chaînes de caractère.

25.2.2 Structures de contrôle

```
if expression :
   instructions
elif expression :
   autres instructions
else :
   instructions finales
```

```
while expression :
   instructions

for variable in range(a,b) ou range(b) ou Liste :
   instructions
```

Les blocs sont décalés de 4 espaces sur la droite après le "**:**"; les "**:**" sont indispensables pour définir les blocs ; on peut définir des séries de données avec des affectations multiples comme

```
>>> a, b, c = 1, 2, 3
```

On peut mettre plusieurs instructions sur une même ligne séparées par des "**;**".

25.2.3 Exceptions

Fréquemment Python renvoie des messages d'erreur dans la console, liés soit à un problème matériel (fichier inexistant, division par 0, etc.), soit à un problème interne (saisie d'un texte alors qu'un nombre est attendu, indice inexistant dans un tableau, etc.).

```
>>> print (-2)**0.5
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
ValueError: negative number cannot be raised to a fractional power
```

Dans tous les cas on peut intercepter l'erreur à l'aide de la séquence **try/except**. Voici un exemple de structure de bloc de gestion d'erreur :

```
instructions
   try:
      instructions : on essaie un truc
   except:
      instructions en cas d'erreur
   else:
      instructions si tout va bien
suite du programme
```

En fait ça fonctionne comme un if. Les instructions d'erreur peuvent être tracées sur plusieurs niveaux.

On peut également obliger Python à provoquer une exception avec l'instruction **raise** :

```
a=1e1000     # Essaye un a négatif puis un a < 1e302
try: b=a**(a**0.5)
except:
    if a<0: raise TypeError ('a est négatif')
    else: raise TypeError ('a est trop gros')
```

Dans cet exemple, on a affiché un message d'erreur dans la console avec **raise**.

25.2.4 Fonctions de base (built_in)

Les fonctions de base qui sont chargées à chaque démarrage de Python sont assez peu nombreuses et concernent essentiellement la manipulation des classes (instructions non listées ici), des listes / tuples / dictionnaires, le typage et transtypage des variables ainsi que quelques fonctions de gestion des chaînes et des fichiers ; il est inutile d'appeler le module **built_in** en début de script.

On se référera pour les détails à la documentation en ligne :

http://docs.python.org/library/functions.html#built-in-funcs

Calculs

`abs(x)`	Valeur absolue (suppression du signe de *x*).	`divmod()`	Renvoie le quotient et le reste dans la div. euclidienne.
`round()`	Arrondi, on peut préciser les décimales.	`pow()`	Puissance, équivalent à x**y.
`eval()`	Renvoie la valeur numérique d'une expression.	`len()`	Longueur de l'argument (liste, chaîne).

Listes

`enumerate()`	Numérote les éléments d'une liste.	`sum()`	Somme des termes d'une liste.
`range()`	Renvoie une liste d'entiers.	`list()`	Retourne une liste à partir d'une suite de termes.
`max()`	Renvoie le maximum d'une liste.	`min()`	Renvoie le minimum d'une liste.
`sorted()`	Renvoie une liste triée.	`slice()`	Renvoie des éléments d'une liste (tranches).
`dict()`	Crée un dictionnaire.	`tuple()`	Convertit une liste en tuple non modifiable.
`zip()`	Combine des listes/tuples.	`map()`	Applique une fonction à tous les éléments d'une liste.
`reversed()`	Renverse une liste.	`next()`	Renvoie l'élément suivant d'un itérateur.

Types

`any()`	Booléen : vrai si au moins un élément d'une liste est vrai.	`all()`	Booléen vrai si tous les éléments d'une liste sont vrais.
`type()`	Type de l'argument.	`bytearray()`	Renvoie un tableau d'octets à partir d'une chaîne par exemple.
`int()`	Convertit une variable en entier, on peut préciser la base.	`bin()`	Convertit un entier en chaîne binaire.
`hex()`	Convertit une variable en hexa (chaîne).	`oct()`	Convertit une variable en base 8 (chaîne).
`float()`	Convertit une variable en réel.	`bool()`	Convertit en booléen.
`complex(x,y)`	Convertit le couple x, y en complexe.	`str()`	Convertit une variable en chaîne de caractères.
`long()`	Convertit une variable en entier long.	`cmp()`	Compare deux variables : – si x<y, 0 si x=y, + si x>y.

Chaînes

`chr()`	Renvoie un caractère dont on connaît le code ASCII.	`ord()`	Renvoie le code ASCII d'un caractère.
`unichr()`	Renvoie un caractère unicode.	`unicode()`	Renvoie le code d'un caractère unicode.
`format()`	Convertit une valeur en chaîne formatée (voir les spécifications).	`basestring()`	String+unicode.
`repr()`	Renvoie une chaîne imprimable.		

Fichiers et modules			
`open()`	Ouvre un fichier.	`execfile()`	Comme **exec** mais sur un fichier.
`print()`	Imprime dans le fichier disponible (la console en standard).	`file()`	Constructeur de fichier (éviter).
`import()`	Importe un module.	`reload()`	Recharge un module déjà chargé.
`input()`	Entrer une valeur au clavier.	`raw_input()`	Entrée au clavier (ce qu'on veut).
`help()`	Appelle l'aide d'un module, d'une fonction.	`dir()`	Renvoie la liste des objets/fonctions d'un module.
`compile()`	Compile en code exécutable.	`vars()`	Dictionnaire des variables.
`id()`	Renvoie l'identificateur interne à Python d'un objet.	`locals()` `/globals()`	Dictionnaire des variables locales / globales d'une fonction /d'un module.

25.2.5 Opérations sur les listes

Quand il y a ***quelque chose**, c'est que **quelque chose** n'est pas indispensable.

Méthodes des listes	Liste vide : **L** = [ ], numérotée à partir de 0 et de longueur **len**(L).
`L.`**`append`**`(élément)`	Ajoute **élément** à la liste **L**. Peut aussi s'écrire **L** = **L** + [**élément**].
`L.`**`insert`**`(index,élément)`	Insère **élément** à la position **index**.
`L.`**`extend`**`(K)`	Ajoute tous les éléments de la liste **K** à la liste **L**. On peut aussi utiliser les opérateurs + et * (à toi de découvrir l'intérêt de *...)
`L.`**`remove`**`(élément)` `L.`**`pop`**`(*index)`	Supprime **élément** de **L** ; erreur si **élément** n'est pas présent. Supprime l'élément n° **index** ; si **index** n'est pas indiqué, le dernier élément est enlevé.
`L.`**`index`**`(élément)`	Indique la position de **élément** ; si pas présent, une erreur se produit.

`L.sort()` `M = sorted(L)`	Tri de **L**. Voir la doc pour les cas spéciaux. Retourne la liste **L** triée dans **M**.
`zip(K, L, M)`	**K**, **L**, **M** sont trois listes de mêmes longueurs ; **zip** fabrique une nouvelle liste constituée de listes contenant les éléments de même index des trois listes. Correspond à $[[K_0,L_0,M_0],[K_1,L_1,M_1],...]$
Référencement : Il est souvent pratique de n'accéder qu'à une partie de la liste **L**.	
`élément in L` `élément not in L`	Booléen disant si **élément** est dans L. Très pratique dans un **if** pour remplacer une série de **or** : au lieu de (x==1) **or** (x==2) **or** (x==3) on mettra x **in** [1, 2, 3].
`L[i : j *:k]`	Renvoie une liste constituée des éléments n° **i** à **j** – 1 de **L** (si **k** est indiqué le compte ira par pas de **k**). Peut servir à insérer une autre liste entre **i** et **j** : **L[i : j]=M**.
`del L[i : j]`	On peut aussi supprimer les éléments avec **L[i : j]** = [ ].
`min(L), max(L), sum(L)`	Évident…

25.2.6 Opérations sur les chaînes de caractères

Fonctions des chaînes	**sa**, **sb** sont des objets chaînes (**str**) : pour obtenir les méthodes disponibles, taper **dir(str)**. Les caractères sont des chaînes de longueur 1.
`sa=chr(n),` `n=ord(sa)`	Renvoie le caractère d'ordre **n** ou le contraire. Lorsqu'on souhaite utiliser des caractères UTF, utiliser **unichr**(n) et **unicode**(sa).
`sa.count(sb` `,*deb,*fin)`	Compte le nombre de fois où **sb** est dans **sa** entre les indices **deb** et **fin**.
`sa.find(sb` `,*deb,*fin)`	Retourne l'indice de la première apparition de **sb** dans **sa** entre les index **deb** et **fin**.
`sa.replace(st_anc` `,st_nouv,*n)`	Remplace les **n** premières apparitions de **st_anc** par **st_nouv**.
`sa.split(sep, *n)`	Sépare **sa** en **n** coupures maximum au niveau de chaque chaîne **sep**.

`sa.`**`lsplit`**`(*char)`	Supprime tous les 'blancs' ou caractère **char** si précisé dans **sa**.
`sa.`**`strip`**`(*s)`	Supprime tous les espaces au début et à la fin de **sa**. Si **s** est donné, tous les caractères faisant partie de **s** sont supprimés de **sa**.
`sa.`**`upper`**`()` `sa.`**`lower`**`()`	Change la casse minuscules vers majuscules ou majuscules vers minuscules de **sa**.
`sa.`**`capitalize`**`()`	Transforme en majuscule le premier caractère de la chaîne **sa**.
`sa.`**`join`**`(Liste_mots)`	Crée une chaîne à partir des mots de **Liste_mots**, séparés par **sa**.
Formatage des nombres	Forme générale : **"%n.df" %x** : n=nombre total de symboles, d=nombre de décimales, f=format de nombre réel, x=nombre auquel s'applique le formatage. Par exemple x=3.14159, **"%05.2f" %x** affiche 003.14 ; on peut manipuler le formatage dans les chaînes de caractère : print St=**"%5.2f" %x** +**' bisous'** affiche **'3.14 bisous'** Voir l'aide de Python pour plus d'infos (fonction **str**).

Formatage des chaînes

Le caractère spécial « \ » (*antislash*) permet quelques subtilités complémentaires d'écriture des chaînes de caractères en Python :

- permet d'écrire sur plusieurs lignes une commande qui serait trop longue pour tenir sur une seule (cela vaut pour n'importe quel type de commande).
- à l'intérieur d'une chaîne de caractères, permet d'insérer un certain nombre de codes spéciaux (sauts à la ligne, apostrophes, guillemets, etc.). Exemples :

```
>>> print '"N\'est-ce pas ?" répondit-elle.'
"N'est-ce pas ?" répondit-elle.
>>> print "Ceci est une chaîne plutôt longue\n contenant plusieurs lignes de
texte (Ceci fonctionne\n de la même façon en C/C++.\n  Notez que les blancs
en début de ligne sont significatifs.\n"
Ceci est une chaîne plutôt longue
contenant plusieurs lignes de texte (Ceci fonctionne
de la même façon en C/C++.
 Note que les blancs en début de ligne sont significatifs.
```

Remarques

- La séquence **\n** dans une chaîne provoque un saut à la ligne.
- La séquence \' permet d'insérer une apostrophe dans une chaîne délimitée par des apostrophes. Même chose avec ", %, t (tabulation).
- Rappelons que la casse est significative dans les noms de variables.
- Pour insérer un \, le répéter ou mettre un **r** devant le texte :

```
>>> s='C:\\fred\\essai.txt'
>>> print s
C:\fred\essai.txt
```

```
>>> s=r'C:\fred\essai.txt'
>>> print s
C:\fred\essai.txt
```

25.2.7 Opérations sur les fichiers

Instructions	Action
`f=`**`open`**`('nomfichier','rwb')`	'**nomfichier**' = nom de fichier valide ; attention aux erreurs d'entrée/sortie (IOError). Paramètres : 'r' = lecture, 'w' = écriture, 'a' pour ajout, 'b' si le fichier est binaire.
`f.`**`read`**`()`	Renvoie TOUT le fichier.
`f.`**`read`**`(x)`	Renvoie les ***x*** premières ligne du fichier.
`f.`**`readline`**`()`	Renvoie la première ligne du fichier.
`f.`**`readlines`**`()`	Renvoie tout le fichier dans une liste, chaque ligne correspond à un élément et finit par "\n".
`f.`**`write`**`('chaîne')` `f.`**`writelines`**`(Liste)`	Écrit <**chaîne**> dans le fichier et ÉCRASE le contenu. Écrit <**Liste**> dans le fichier sans retour entre les lignes.
`f.`**`close`**`()`	Fermeture du fichier et validation des écritures.

25.3 Les fonctions

Une fonction est un morceau de code plus ou moins autonome servant soit à alléger le code principal, soit plus généralement à effectuer une opération répétitive. Une fonction est déclarée par le mot clef **def** et son nom se termine par :, de sorte que le code de la fonction est incrémenté.

```
def bonjour:
    print 'bonjour'
```

`bonjour()`

ce n'est pas pareil que `bonjour` sans ()

On peut passer des **paramètres** à une fonction, c'est-à-dire des valeurs fixes ou des variables qui vont permettre à la fonction de s'exécuter :

```
def bonjour(prenom) :
    print 'bonjour', prenom
```

`bonjour('Fred')` affiche `bonjour Fred`

mais on peut aussi faire

`petitnom='Fred';bonjour(petitnom)`

On peut fournir une valeur **par défaut** à nom :

```
def bonjour(prenom='Fred') :
    print 'bonjour', prenom
```

`bonjour()` affiche `bonjour Fred`

Plusieurs paramètres peuvent être passés avec des valeurs par défaut ou pas :

```
def bonjour(nom, prenom='Fred') :
    print 'bonjour', prenom, nom
```

`bonjour('Laroche')` affiche

`bonjour Fred Laroche`

On peut utiliser le nom d'un paramètre comme nom de variable :

```
def bonjour(nom, prenom='Fred') :
    print 'bonjour', prenom, nom
```

`nom='Laroche'`
`bonjour(nom)`
affiche `bonjour Fred Laroche`
`nom='Laroche'`
`bonjour(nom,'Lucie')`
affiche `bonjour Lucie Laroche`

Certaines variables peuvent être utilisées uniquement à l'intérieur du code d'une fonction (on dit qu'elles sont **locales**), d'autres variables peuvent exister en dehors de la fonction, on dit qu'elle sont **globales**. Si on utilise une variable globale, il faut le dire :

```
nom='Laroche'
def bonjour(prenom='Fred') :
    global nom
    age=45
    print 'bonjour', prenom, nom, 'tu as',2*age,'ans'
```

`bonjour()`

affiche

`bonjour Fred Laroche tu as 90 ans`

La variable **nom** peut être réutilisée après la fonction **bonjour**.

prenom est un paramètre fixe qui ne peut pas être modifié dans la fonction.

La variable **age** est manipulable à l'intérieur de la fonction **bonjour** mais disparaît une fois que **bonjour** a été exécutée.

D'une manière générale on essaie d'utiliser un minimum de variables globales.

```
def bonjour(nom, prenom) :
    return 'bonjour '+prenom+' '+nom
print bonjour('Laroche','Fred')
```

Le résultat de `bonjour(…)` est une chaîne que l'on peut afficher.

Très souvent une fonction retournera une ou plusieurs valeurs :

```
def bonjour() :
    prenom=raw_input('Prenom ?')
    age=raw_input('Age ?')
    return prenom, age

z=bonjour()
print z[0]+' a '+z[1]+' ans.'
```

Le résultat de `bonjour()` est une liste constituée de deux chaînes que l'on peut afficher.

Question : pourquoi utilise-t-on ici la variable intermédiaire **z** ?

La plupart des instructions qui suivent sont étudiées dans le livre :

25.4 Tkinter

25.4.1 Forme générale

`from Tkinter import *`			En début d'application on charge la bibliothèque Tkinter.
`fenetre=Tk()`			On définit une instance de la classe Tk.
w_1=**Label**(**args*)	*w_2*=**Button**(**args*)	*w_3*=**Canvas**(**args*)…	Création de widgets.
w_k.**grid**() (ou *w_k*.**pack**() pas recommandé) *w_k*.**place**(*x*, *y*) *w_k*.**config**(*position*, *couleurs*, *police*, *texte*, …)			Positionnement des widgets. Configuration des widgets si pas déjà fait.

Dans le widget Canvas (ci-dessous nommé *w*) on peut fabriquer des objets :

a=*w*.**create_line**()	*b*=*w*.**create_oval**()	*c*=*w*.**create_poly**(tags='*fred*', …)	Création des objets du Canvas

Une fois un objet créé il reçoit un identificateur **id** (un entier) contenu dans la variable *a*, *b* ou *c* ; on peut également lui affecter un **tag** (étiquette) ; plusieurs objets peuvent avoir le même **tag** (pratique pour manipuler un groupe d'objets). On peut alors faire des modifications sur les objets à partir de leur **id** ou de leur **tag**.

w.**coords**(*c*, x1, y1, x2, y2, x3, y3, x4, y4) ou *w*.**coords**('*fred*', x1, y1, x2, y2, x3, y3, x4, y4) *w*.**itemconfigure**('*fred*', fill=couleurfond, outline=couleurbord, stipple=transparence)	Modification des objets ayant l'id *c* ou le tag *fred.*

Quand tout est dessiné, on attend les événements qui vont se produire…, événements liés à un widget.

w.**bind**(*evenement*, *fonction*) #event n'est pas passé en paramètre dans bind **def** *fonction*(event) #mais doit être passé dans la définition de la fonction. *instructions*	On lie l'*evenement* qui se produit dans le widget *w* à *fonction ;* définition de la fonction concernée. Il y aura diverses fonctions suivant ce qui se passe.
fenetre.**mainloop()**	Gestionnaire événements

Autres instructions comme **fenetre.quit**() pour sortir proprement. Enjoy !

25.4.2 Widgets

(abréviation pour Window Gadget…)

`Button`	Un « simple » bouton, utilisé pour exécuter une commande ou autre opération.
`Canvas`	Graphique structuré : utilisé pour dessiner des figures ou des graphes ainsi que des widgets personnalisés. Nécessaire pour dessiner facilement.
`Checkbutton`	Bascule oui/non. Nécessite une variable spéciale (IntVar).
`Entry`	Champ d'entrée de texte. Nécessite une variable spéciale (StringVar).
`Frame`	Widget de contenu : peut avoir une bordure, un fond ; utilisé pour regrouper d'autres widgets comme des boites de contrôle.
`Label`	Affiche un texte ou une image.
`Listbox`	Affiche une liste de choix : peut-être configurée en bouton-radio.
`Menu`	Panneau de menu. Utilisé pour plusieurs types de menus.
`Menubutton`	Bouton de menu déroulant.
`Message`	Affiche un texte : semblable au Label mais peut s'adapter automatiquement à une taille.
`Radiobutton`	Représente une valeur d'une variable pouvant en prendre plusieurs. Une fois sélectionné réinitialise tous les boutons liés à cette variable.
`Scale`	Permet de fixer la valeur d'une variable avec un curseur.
`Scrollbar`	Barre de défilement standard (utiliser avec canvas, entry, listbox, text).
`Spinbox`	Permet de choisir la valeur d'une variable dans un jeu donné de nombres ou un jeu fixe de chaînes. Voir ici : http://effbot.org/tkinterbook/spinbox.htm
`Text`	Affiche un texte formaté, permet divers styles et attributs. Supporte les images.
`Toplevel`	Widget de contenu qui s'affiche comme une fenêtre chapeau.

Note : de nombreux widgets utilisent des classes de variables particulières comme IntVar, StringVar, FloatVar…; par exemple le widget Checkbutton va utiliser un IntVar à déclarer comme suit :

```
u=IntVar(nom_fenetre,value=valeur_entiere_initiale)
```

On récupère / définit les données (ici un entier) avec une instruction du type :

```
u.get() / u.set()
```

25.4.3 Méthodes

`grid` `grid_forget`	Permet de créer une grille dont chaque case peut recevoir un objet, très utile si on a plusieurs widgets à placer. Options pour un objet : *column*, *columnspan*, *row*, *rowspan*, *sticky*. *Forget* permet d'effacer le widget à l'écran.
`pack` `pack_forget`	Empaquetage des widgets dans un widget (ou la fenêtre) les uns au dessus des autres. Avec un seul widget permet d'actualiser l'affichage. 3 options : *side*, *expand*, *fill* (110)
`place` `place_forget`	Donne un emplacement explicite (coordonnées écran). Peut être utilisée avec **grid** ou **pack**. Attention par contre à ne pas mélanger **grid** et **pack** dans la même application.
`configure`	Commande permettant de modifier des options de n'importe quel widget déjà placé sur l'interface. Exemple : pour changer la couleur d'un label, *nomwidget*.configure(bg="blue",fg="white")
`bind`	*widget*.bind(*evenement*, *fonction*), fonction est la fonction appelée lorsque *ev.* survient. Exemple : *b*=**Button**(**text**="Appuyer sur une touche") #définition du widget. **def** affiche_touche(evt) : # fonction à activer. **print** 'evt.char = ', evt.**keysym** # affiche la touche pressée. *b*.**bind**('<Key>', affiche_touche) #pas de paramètre à affiche_touche. *b*.**focus_set**() # donne la main au bouton b.

25.4.4 Classes spéciales

`BitMapImage`	Voir PIL pour les transferts.
`Font`	Gestion des polices de caractère.
`PhotoImage`	Voir PIL pour les transferts.

25.4.5 Gestion des événements

`<Button-i>`	Pression d'un bouton de la souris, i=1, 2, 3.
`<ButtonRelease-i>`	Relâchement du bouton i.
`<Double-Button-i>`	Double click du bouton i.

`<Motion>`	Mouvement de la souris.
`<Enter> / <Leave>`	La souris entre / quitte dans la zone de l'objet.
`<Key>`	N'importe quelle touche (y compris les flèches et autres touches muettes).

25.4.6 Contenu de la variable *evenement* (evt)

`char`	Code de la touche pressée : caractères imprimables.
`keysym`	Code de la touche pressée : toutes sont admises.
`num`	Numéro du bouton de souris appuyé.
`x, y`	Coordonnées de la souris par rapport au coin sup. gauche de l'objet.
`x_root, y_root`	Coordonnées absolues de la souris.
`widget`	Identifiant (adresse mémoire) permettant d'accéder à l'objet ayant reçu l'évt.

25.4.7 Dessiner dans un Canvas

`create_arc`	Portion d'ellipse.
`create_bitmap`	Image bitmap en noir et blanc (voir plus haut).
`create_image`	Image graphique *gif* (voir plus haut).
`create_line`	Une ou plusieurs lignes.
`create_oval`	Une ellipse, utilisée également pour les cercles.
`create_polygon`	Polygone : on donne les coordonnées de tous les points.
`create_rectangle`	Rectangle.
`create_text`	Dessiner un texte, différent de Label.
`create_window`	Une fenêtre rectangulaire (différent de Tk()).