Programmation avec Python

Université Nice SOPHIA ANTIPOLIS

Pourquoi Python ?

Pour programmer, il nous faut un langage. Il en existe de très nombreux, chacun étant plus ou moins facile à aborder, chacun plus ou moins adapté à tel ou tel domaine. Le langage python présente de nombreux avantages:

- La syntaxe du langage incite à la clarté, ce qui facilite grandement la relecture des programmes ; il est en effet très pénible de « reprendre » un programme de quelqu'un d'autre, ou que l'on a écrit soi-même longtemps auparavant, dont le simple aspect visuel est embrouillé

- C'est un langage orienté objet, actuellement le paradigme de programmation le plus répandu (mais il sera essentiellement utilisé en tant que langage procédural durant ce cours...)

- C'est un langage interprété, donc sa mise en oeuvre est très simple
l'interpréteur, le logiciel qui permet à vos programmes de s'exécuter, est un logiciel libre, gratuit, et multi-plateforme : un programme écrit sous Linux peut fonctionner sous Windows sans aucune modification

- Langage moderne et développement très rapide

- Il est assez plaisant car de « haut niveau ».

Pourquoi Python ?

Il est nécessaire d'installer l'interpréteur python sur votre ordinateur. La procédure à suivre diffère selon votre système d'exploitation

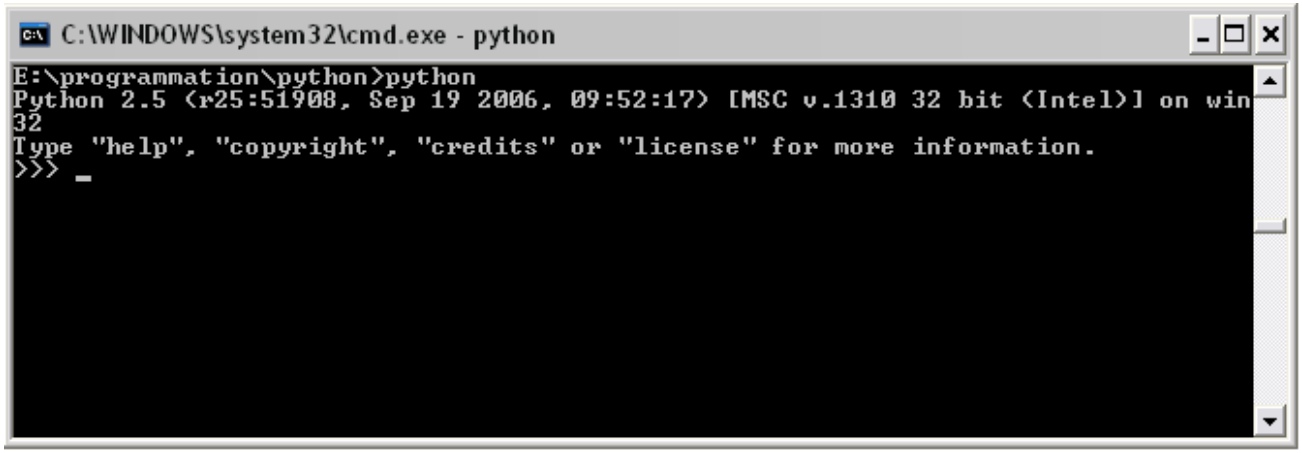
Sous Linux il est très souvent installé par défaut.

Sous Windows l'installation est très facile (consultez le site Web de Python pour télécharger le programme d'installation)

Il est recommandé d'installer TKinter (bibliothèque graphique) pour profiter pleinement de python. (www.activestate.com)

Premières instructions avec Python

Le coeur d'un langage interprété, c'est l'interpréteur.
Pour le faire fonctionner ouvrez un terminal (« cmd » sous windows), puis lancez la commande python.
Note: si la commande n'est pas au « path » il faut la lancer depuis l'emplacement de l'installation



Les caractères >>> à gauche sont le prompt de l'interpréteur Python, qui vous signale qu'il est prêt à recevoir vos instructions.

Essayer (en appyant sur Entrée à chaque fin de ligne):

```
2+3
5.95 *  6.55957
print "Bonjour !"
```

Premières instructions avec Python

Ces exemples simples nous donnent déjà plusieurs informations :

- on peut utiliser l'interpréteur Python comme une calculatrice ;
- pour effectuer une addition, on l'écrit naturellement, en utilisant l'opérateur + ;
- les nombres décimaux sont notés à l'anglaise, en utilisant le point plutôt que notre virgule ;
- la multiplication est effectuée par l'opérateur * (l'étoile, ou astérisque) ;

remarquez que l'on peut insérer ou non à loisir des espaces entre les valeurs et l'opérateur.
les chaînes de caractères, sont délimitées par le caractère guillemet " (ou double-quote)

<u>Les variables:</u>

```
x = 5
```

Pour réutiliser cette variable, il suffit d'inscrire son nom, par exemple :

```
2*x
```

Ce qui donne : 10

on peut redéfinir la variable:

```
x = 6
x = 2*x
```

Ce qui donne : 12

Premières instructions avec Python

Les variables peuvent naturellement être utilisées pour stocker d'autres choses que des nombres entiers, et les noms ne se limitent pas à une lettre :

```
pi = 3.14159
formation = "Licence DORA"
```

En langage Python, comme dans la plupart des langages, les noms de variables peuvent être composés des lettres de l'alphabet, des chiffres et du caractères de soulignement _ (underscore), avec comme contrainte que le premier caractère ne doit pas être un chiffre. Par ailleurs, Python distingue les majuscules des minuscules : x et X peuvent être deux noms de variables différentes. D'une manière générale, il est vivement recommandé de donner des noms explicites aux variables.

Dans les exemples précédents, x était une variable de type entier, pi de type réel (ou virgule flottante), « formation » de type chaîne de caractères. Certains langages, comme Python, déterminent le type d'une variable lors de sa déclaration (qui se trouve être également une affectation). D'autres, comme C/C++, imposent de définir explicitement ce type avant toute affectation.

Faire un code source

L'utilisation de l'interpréteur telle que nous l'avons fait jusqu'ici ne permet pas de construire un programme complexe, simplement parce qu'il faudrait toujours retaper les instructions. C'est pourquoi les instructions d'un programme sont stockées dans un ou plusieurs fichiers, l'ensemble de ces fichiers constituant le code source : le code tapé par le programmeur pour obtenir le programme désiré.
Pour créer un code source, il suffit d'un éditeur de texte. Attention, il s'agit bien de texte brut, sans aucune mise en forme donc un éditeur simple est suffisant.

Scite (= www.scintilla.org) est diponible sous Windows et Linux. Il est simple est convient très bien pour coder en python ou dans un autre langage.

Ouvrez Scite (ou le un autre éditeur de texte vous préférez) et taper:

```
print "Bonjour !"
str = "x vaut"
x = 5
print str, x
```

sauvegardez votre fichier avec l'extention .py

Faire un code source

Votre programme doit être interprété pour être exécuté:
Pour l'interpréter, à partir d'une ligne de commande, lancez « python prog.py »

L'installation de python sous Windows associe en général les fichiers .py avec l'interpréteur python. Il suffit alors de double-cliquer directement sur le fichier dans l'explorateur.

Avec Scite il est possible de lancer directement l'interprétation du script en cours en appuyant sur F5. Le résultat s'affiche alors dans une fenêtre annexe.

Manipulation de variables

Généralité sur les variables

une variable possède un type, qui précise la nature de l'information qu'elle peut contenir.
Testez:

```
x = 1
print type(x)
```

type() est une fonction (rassemble des instructions). Ici le rôle de cette fonction est de donner le type de ce qui se trouve entre les parenthèses (on dit aussi, son paramètre).

Testez:

```
x = 3.14
print type(x)
```

Manipulation de variables

Conversions entre types

Effectuer une conversion de type consiste à consulter le contenu d'une variable comme si elle était d'un type différent. Par exemple, créons une variable chaîne de caractères : >>> s = "10"

Cette chaîne représente une valeur numérique, mais pour l'instant ce n'est qu'une chaîne.
Ajouter une valeur
print s + 5 Que ce passe-t-il ?

La solution consiste à effectuer une conversion, dans notre cas à transformer la chaîne de caractères en une valeur numérique correspondante :

```
print int(s) + 5
```

int() est une fonction qui tente de fabriquer un entier à partir de ce qu'on lui donne entre parenthèses,
Testez:

```
s = "erreur !"
print int(s) + 5
```

Puis:

```
print int(10.8) + 5
```

Manipulation de variables

Testez:

```
 s = "10.7"
 x = float(s) + 1.1
print x

 print str(x) + "0"
```

Résultat?

Les chaînes de caractères

Les chaînes de caractères sont en fait une suite de caractères en mémoire, que l'on peut représenter sous la forme d'un tableau

Il est possible d'obtenir un élément individuel de cette zone, c'est-à-dire un caractère individuel :

```
print s[2]
c
```

Manipulation de variables

La chaîne est considérée comme un tableau. L'utilisation des crochets nous permet d'indiquer quel élément du tableau nous voulons obtenir. Le nombre donné entre les crochets est appelé l'indice de l'élément voulu. Ici, nous demandons l'élément d'indice 2. Par convention, les indices commencent à 0
Il est possible de donner des indices négatifs, la numérotation commence alors par la fin :

```
print s[-2]
d
```

Découpage

```
print s[1:4]
bcd

print s[1:-2]
bc
```

Si le premier membre manque, cela signifie « depuis le début ». Si le deuxième manque, cela signifie « jusqu'à la fin ». Par exemple :

```
print s[2:]
cde
```

Manipulation de variables

Assemblage (=concaténation »)

```
s1 = "bonjour "
s2 = "monde !"
s = s1 + s2
print s
```

La chaîne s est le résultat de la concaténation des chaînes s1 et s2. En Python, la concaténation est réalisée par l'opérateur d'addition + (si les membres sont des chaînes !)

```
s1 += s2
print s1
bonjour monde !
```

+= combine en une seule opération la concaténation puis l'affectation dans la chaîne donnée à sa gauche. La ligne s1 += s2 est donc équivalente à s1 = s1 + s2, c'est-à-dire, « coller les chaînes s1 et s2, puis stocker le résultat dans s1 ».

Manipulation de variables

Exercice:

Soit s la chaîne « Bonjour monde! »

-Faites un script qui permet d'insérer la chaîne « tous le » dans s entre « bonjour » et « monde »

- afficher s

- Corriger le « tous » en « tout » et réafficher la chaîne

Autre astuce sur les chaînes: il est possible de répéter un chaîne en multipliant la chaîne par un entier

Exemple:

```
 s = "toto " * 5
 print s
```

toto toto toto toto toto

Manipulation de variables

Opérations arithmétiques

```
 a = 5
a += 2
print a
7
```

```
>>> 2**128
340282366920938463463374607431768211456
>>> 2.0**128
3.4028236692093846e+38
```

Mathématiquement, « 2 » et « 2.0 » sont parfaitement équivalents, mais l'ordinateur ne les stocke pas du tout de la même manière. Python nous renvoie un résultat exact dans le premier cas (39 chiffres), mais un résultat approché dans le deuxième (« seulement » 17 chiffres).

-> Les nombres décimaux ont une précision limitée. En général, dès que l'écriture du nombre nécessite plus de 16 chiffres, l'ordinateur effectue un arrondi et ne manipule plus qu'une valeur approximative, d'où un certain risque d'erreur pour des calculs importants

```
10.0**50 + 2 - 10.0**50
0.0
```

Attention !!!

Les fonctions

Les fonctions sont des mini-programmes qui évitent de répéter une séquence d'instructions.
Une fonction est donc une suite d'instructions qui peut être exécutée à la demande par un programme : tout se passe alors comme si on avait inséré les instructions de la fonction à l'endroit où on l'appelle.

Définir une fonction

```
def Ma_fonction () :
        instruction1
        instruction2


instruction 1, programme principal
Ma_fonction()
instruction 3, programme principal
...
```

le mot-clé def : c'est par lui que l'on informe Python que ce qui suit est la définition d'une fonction ;
ensuite, le nom de la fonction, ici « Ma_fonction »;
puis une paire de parenthèses ; elles sont nécessaires, elles contiennent éventuellement des arguments ;
Les deux-points, qui indiquent à Python que l'on s'apprête à donner les instructions contenues dans la fonction.

Les fonctions

Les instructions contenues par exemple dans une fonction constituent **un bloc d'instructions**
En python un bloc est défini par une **indentation** similaire pour une suite d'instructions (tabulations, espaces, etc.)

Selon les langages, différentes techniques sont utilisées pour repérer le début et la fin d'un bloc : par exemple, en langages Ada ou Pascal, le début est marqué par begin et la fin par end. En langage C/C++, php, perl on utilise les accolades { et }

A la fin de chaque ligne d'instructions, il n'y a aucun caractère contrairement à d'autres langages où chaque instruction se termine par un ;

Les lignes vides ne sont pas interprétées et servent simplement à rendre le code plus lisible

Les fonctions

L'appel d'une fonction

Pour appeler une fonction, il suffit d'inscrire son nom, avec une paire de parenthèses qui contient des arguments s'ils existent. Il faut en outre avoir défini la fonction avant son appel.

Exercice: refaire le code suivant en utilisant une fonction

```
print "ligne 1"
print "_"*10
print "-"*10
print "ligne 2"
print "_"*10
print "-"*10
print "ligne 3"
print "_"*10
print "-"*10
print "ligne 4"
```

Les fonctions

Exercice (correction):

```
def ligne():
        print "_"*10
        print "-"*10

print "ligne 1"
ligne()
print "ligne 2"
ligne()
print "ligne 3"
ligne()
print "ligne 4"
```

Les fonctions

Les paramètres de fonctions (=arguments)

Les paramètres d'une fonction la rendent plus générique, c'est-à-dire qu'elle pourra adapter son action selon la situation.
Les paramètres sont transmis à la fonction lors d'appel. Lors de l'appel il peut s'agir de variables ou directement de valeurs. Si c'est une variable, python la remplace par sa valeur.
Au niveau de la fonction la ou les valeurs sont envoyées dans des variables (qu'il faudra nommer dans les parenthèses)

```
def Ma_fonction (param) :
        instruction1
        instruction2

instruction 1, programme principal
Ma_fonction(valeur)
instruction 3, programme principal
```

Les variables dans une fonction sont « locales » c'est à dire qu'elles n'existent que dans la fonction, le programme principal ne les connaît pas.

Les fonctions

Il est possible de passer plusieurs paramètres à une fonction.

Les paramètres de la fonction sont récupérées dans le même ordre que lors de l'appel.

Exemple:

```
def Ma_fonction (param1, param2) :
        instruction1
        instruction2
instruction 1, programme principal
Ma_fonction(valeur1, valeur2)
```

param1 = valeur1

param2 = valeur2

Il est cependant possible de passer les arguments dans le désordre si les valeurs passées lors de l'appel sont « nommées ».

```
def Ma_fonction (param1, param2) :
        instruction1
        instruction2
instructions 1 programme principal
Ma_fonction(param2=valeur2, param1=valeur1)
```

Les fonctions

Il est possible de spécifier, dans la définition de la fonction, une **valeur par défaut** pour un ou plusieurs paramètres, c'est-à-dire la valeur qu'ils auront s'ils ne sont pas présents à l'appel de la fonction.

```
def Ma_fonction (param1="toto", param2="tata") :
    instruction1
    instruction2


instruction 1, programme principale
Ma_fonction(param1=valeur1)
```

param1 = valeur1
param2 = "tata"

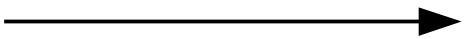
En revanche, si vous ne spécifiez pas de valeur par défaut et que vous mettez moins de paramètres lors de l'appel que dans la fonction, il y aura une erreur.

Les fonctions

Exercice:

Modifier le programme précédent pour que l'on puisse spécifier lors de l'appel de la fonction:
- le caractère à afficher (caractères _ et – dans le programme actuel). Vous pouvez mettre les caractères que vous voulez...
- le nombre de fois que le caractère doit être répété (dans le programme actuel, les caractères sont répétés 10 fois) Appeler la fonction avec la valeur 10, puis 20, puis 30.

```
def ligne():
        print "_"*10
        print "-"*10

print "ligne 1"
ligne()
print "ligne 2"
ligne()
print "ligne 3"
ligne()
print "ligne 4"
```



?

Les fonctions

Exercice (correction possible)

```
def ligne(car="-", nb=10):
        print car*nb

print "ligne1"
ligne()
ligne("$",10)
print "ligne2"
ligne("-",20)
ligne("*",20)
print "ligne3"
ligne("-",30)
ligne("@",30)
print "ligne4"
```

```
ligne1
----------
$$$$$$$$$$
ligne2
--------------------
********************
ligne3
------------------------------
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
ligne4
```

Les retours de fonctions

La possibilité de définir des fonctions est un grand principe de la **programmation structurée**. C'est-à-dire que l'on s'efforce d'organiser les programmes selon une méthode consistant à diviser les gros problèmes en plus petits. On constitue ainsi des éléments de programmes, qui communiquent et travaillent ensemble, tout en évitant dans la mesure du possible que ces éléments dépendent trop les uns des autres.
L'objectif étant d'éviter le fouillis que pouvaient être d'anciens programmes écrits par exemple en langage Basic.
Dans le programme précédent nous avons une fonction qui effectue un calcul, à savoir construire une chaîne de caractères et qui l'affiche. Or dans un « bon » programme , une fonction n'a pas à «savoir» ce que l'on va faire de cette chaîne de caractères, elle doit juste effectuer le calcul.

Par exemple comment faire si, demain, au lieu d'afficher cette chaîne nous voulons l'écrire dans un fichier ? Il faudrait modifier la fonction, c'est ce qu'il faut éviter...

Les fonctions

L'idée consiste donc à définir des fonctions qui font ce pour quoi elles ont été créées, mais pas plus. Notre fonction calcule une chaîne de caractères mais ce n'est pas son rôle de l'afficher. Dans notre cas, ce rôle est dévolu au programme principal. Il nous faut donc un moyen pour « récupérer » ce que la fonction aura produit. On dit alors que la fonction va **retourner un résultat**, ce résultat étant la **valeur de retour** de la fonction. Ce que nous ferons de ce résultat est la responsabilité de celui qui appelle la fonction, dans notre cas le programme principal.

Exemple:

```
def ma_fonction (var1, var2) :
    resultat = var1 + var2
    return resultat

ma_variable = ma_fonction(2,3)
print ma_variable
```

- « resultat » est une **variable locale** (interne à la fonction et inconnue par le programme principal
- « return resultat » signifie que la fonction **renvoie** au programme principal le contenu de la variable « resultat »
- Dans le programme principal la **valeur de retour** de la fonction est stockée dans la variable «ma_variable»

Interactions avec le programme

1 - Création d'un jeu simple: inter-agir avec l'utilisateur

Par « contrôler ou interagir avec le déroulement d'un programme », on entend généralement diriger la succession des instructions, par exemple choisir entre certaines séquences selon divers critères, ou provoquer la répétition de l'exécution d'une certaine séquence.

Pour illustrer ces principes, nous allons réaliser le jeu suivant : un programme doit choisir un nombre au hasard entre 1 et 10, et l'utilisateur devra deviner ce nombre en faisant des propositions. À chaque proposition, le programme affichera « trop grand » ou « trop petit », selon le cas.

```
from random import randint

nombre_hasard = randint(1, 10)

chaine = raw_input("Donnez un nombre : ")

nombre_joueur = int(chaine)
```

1- Cette ligne donne accès à une fonction nommée randint(), qui permet d'obtenir un nombre au hasard compris entre deux nombres donnés en paramètre. Elle est contenue dans un « module » (=ensemble de fonctions similaires regroupées)

2 - Le résultat de la fonction randint() est stocké dans la variable nombre_hasard. C'est ce nombre que l'utilisateur devra trouver par essais successifs.

3 - La fonction raw_input() est en quelque sorte l'opposée de la commande print, que nous avons déjà beaucoup utilisée. Son rôle est d'attendre que l'utilisateur entre quelque chose au clavier, la chaîne de caractères qui est passée en paramètre étant affichée en début de ligne

4 - raw_input() renvoie toujours une chaîne de caractères, or nous voulons un nombre. Il faut donc transformer cette chaîne en nombre, qui sera stocké dans «nombre_joueur».

Interactions avec le programme

2 - Création d'un jeu simple: De l'algorithme au programme

- Après que l'utilisateur ait saisi son nombre il faut afficher une réponse qui tient compte du nombre de l'utilisateur et du nombre aléatoire créé par notre programme. Il va donc y avoir une « comparaison » de variables.

En oubliant pour l'instant le cas où le nombre l'utilisateur trouve exactement le nombre aléatoire, notre algorithme peut se décomposer de la manière suivante:

*si le nombre donné est plus grand que le nombre de référence, alors le message est "**Trop grand**", sinon le message est "**Trop petit**".*

- Le but est de s'approcher du langage Python, en remplaçant certains termes par les variables qu'ils représentent :

*si **nombre_joueur** est plus grand que **nombre_hasard**, alors afficher etc...*

La notion de message, dans notre cas, correspond à un affichage à l'écran, ce que permet la commande print. Par ailleurs, la comparaison de deux nombres se fait en utilisant les opérateurs mathématiques usuels que sont les symboles < et >.

Interactions avec le programme

•*si* ***nombre_joueur*** > ***nombre_hasard***, *alors* ***print "Trop grand"***, *sinon* ***print "Trop petit"***.

En python le « si » est représenté par « if » **If (condition) : (instructions à exécuter)**

Le « alors » est représenté par les « **:** »

Le « sinon » est représenté par « else » **else : (instructions à exécuter)**

Cela donne si on reprend notre algorithme précédant:

```
 if ( nombre_joueur > nombre_hasard ) :
        print "Trop grand !"
else :
        print "Trop petit !"
```

Notez que les instructions sont rattachées à leur condition par des indentations.

Les parenthèses pour le if sont facultatives mais il vaut mieux prendre l'habitude de les mettre.

Le « else » n'est évidemment pas obligatoire s'il n'y a qu'une seule alternative possible.

Maintenant modifier le programme pour tenir compte du cas de l'égalité entre les 2 nombres.

L'opérateur de comparaison pour les égalités se note « == » (double égal)

Interactions avec le programme

```
if ( nombre_joueur == nombre_hasard ) :
        print "Vous avez trouvé!"
 else :
        if ( nombre_jouer > nombre_hasard ) :
                print "Trop grand!"
        else :
                print "Trop petit!"
```

Il y a ici une **imbrication de blocs.**

Notre programme n'est pas terminé : actuellement l'utilisateur ne peut proposer qu'un seul nombre, puis le programme se termine. Comment faire pour que l'utilisateur puisse proposer plusieurs nombres ? Une solution consisterait à dupliquer les lignes de code. Mais cela ne paraît pas une bonne solution, surtout si on se trouve confronter à des centaines d'instructions plutôt que quelques-unes.

Maintenant, nous voulons que l'utilisateur tente plusieurs valeurs pour trouver le nombre aléatoire

Interactions avec le programme

3 - Création d'un jeu simple: introduction aux structures répétitives (=boucles)

tant que les nombres ne sont pas égaux, afficher un message adapté puis demander un nouveau nombre.

La partie « afficher un message » a déjà été traitée, nous pouvons donc remplacer ce membre par notre principe précédent (que nous savons déjà traduire) :

*tant que les nombres ne sont pas égaux, si le nombre donné est plus grand que le nombre de référence, alors afficher "**Trop grand**", sinon afficher "**Trop petit**", puis demander un nouveau nombre.*

Le mot « tant que » sous-entend que nous allons effectuer les opérations qui suivent jusqu'à ce qu'une certaine condition soit vérifiée, ou plus précisément ici, tant qu'une condition n'est pas vérifiée. Premier problème, comment traduire l'expression « ne sont pas égaux » ? Il suffit d'exprimer la négation de la condition «sont égaux». En langage Python, la négation d'une condition s'écrit simplement **not**. En effectuant les remplacements par les variables que nous avons, et en utilisant les opérateurs de comparaison, nous pouvons réécrire la phrase ainsi :

*tant que **not** (**nombre_joueur** == **nombre_hasard**), si (**nombre_joueur** > **nombre_hasard**) alors **print "Trop grand"**, sinon **print "Trop petit"**, puis demander un nouveau nombre.*

Sachant que « tant que » se traduit en python par « while », terminer le programme...

Interactions avec le programme

Le programme complet:

```
from random import randint
nombre_hasard = randint(1, 10)
chaine = raw_input('Donnez un nombre : ')
nombre_joueur = int(chaine)

while ( not  (nombre_joueur == nombre_hasard) ) :

        if ( nombre_joueur > nombre_hasard ) :
                print "Trop grand !"
        else :
                print "Trop petit !"

        chaine = raw_input("Donnez un nombre : ")
        nombre_joueur = int(chaine)

print "bravo !"
```

Note: La condition de la boucle peut également être écrite de cette façon:

while (nombre_joueur != nombre_hasard) :

L'opérateur « != » signifie « est différent »

Interactions avec le programme

3 notions à retenir:

- La notion d'alternative, qui permet d'orienter le flux d'un programme. Grâce à cela, certaines parties d'un programme ne seront exécutées que selon certaines conditions données.

- La notion de boucle, qui permet d'exécuter plusieurs fois une même suite d'instructions. La fin de la boucle, dépend d'une condition donnée.

- La notion de condition qui est implicite des 2 premières. La valeur d'une condition est donnée par le résultat d'un calcul particulier, qui est le test correspondant à la condition. À strictement parler, le type de cette valeur n'est ni un entier, ni une chaîne de caractères, il s'agit d'un type dit booléen (vrai ou faux, 0 ou 1)

Ces trois notions font parties des fondements de l'algorithmique. Pour rappel, un algorithme est la description d'une suite d'actions ou d'opérations à effectuer dans un certain ordre. Au cours de cette suite peuvent intervenir des boucles ou des alternatives.

Exercice récapitulatif: reprendre le dernier programme et le modifier de la façon suivante:

- Créer une fonction qui demande le nombre à l'utilisateur et retourne ce nombre au programme principal (la fonction est donc appelée 2 fois dans le programme principal et remplace 4 instructions)

- Après avoir trouvé le nombre, le programme doit proposer à l'utilisateur de rejouer:

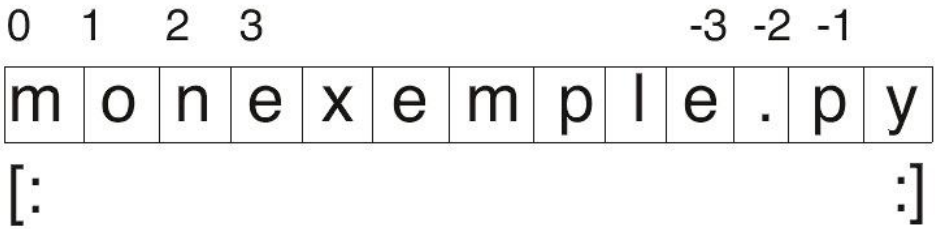
Si l'utilisateur tape « oui » , le jeu est relancé , si « non » ou autre chose, le programme se termine

Interactions avec le programme

```
from random import randint

def nombre ():
	chaine = raw_input("Donnez un nombre : ")
	chaine = int(chaine)
	return chaine

jouer = "oui"

while (jouer == "oui") :

	nombre_hasard = randint(1, 10)
	nombre_joueur = nombre()

	while (nombre_joueur != nombre_hasard)  :

		if ( nombre_joueur > nombre_hasard ) :
			print "Trop grand !"
		else :
			print "Trop petit !"

		nombre_joueur = nombre()

	jouer= raw_input("Bravo!\nVoulez-vous rejouer ? ")
```

Manipulation de variables (suite)

Rappel sur les variables:
- les variables simples: entiers, réels, etc.
- les chaînes



Les listes

La liste est l'objet de collection ordonné le plus souple. Il permet de contenir toutes sortes d'objets : nombres, chaînes, et même d'autres listes. La déclaration d'une liste se fait par la mise entre crochets des objets de celle-ci.
x = [0, 1, «hello»]

Du point de vue de l'utilisation, les listes permettent de collecter de nouveaux objets afin de les traiter tous ensembles.
Comme les chaînes, les objets d'une liste sont sélectionnables par indiçage. Les listes sont ordonnées et autorisent donc l'extraction de tranches et la concaténation.
Les listes permettent une modification de leur contenu : remplacement, destruction,... d'objets sans pour autant devoir en faire une copie (comme les chaînes). On dit quelles peuvent êtres changées sur place.

Manipulation de variables (suite)

Opération	Interprétation
L1=[]	liste vide
L2=[0, 1, 2, 3]	4 élément indicé de 0 à 3
L3=['abc', ['def', 'ghi']]	liste incluses
L2[i], L3[i][j]	indice
L2[i:j]	tranche
len(L2)	longueur
L1+L2	concaténation
L1*3	répétition
for x in L2	parcours
3 in L2	appartenance
L2.append(4)	méthodes : agrandissement
L2.sort()	tri
L2.index()	recherche
L2.reverse()	inversion
del L2[k], L2[i:j]=[]	effacement
L2[i]=1	affectation par indice
L2[i:j]=[4, 5, 6]	affectation par tranche
range(4), xrange(0,4)	céation de listes / tuples d'entiers

- Les listes emboîtées constituent des listes à n dimensions. Ce sont des matrices (ou tableaux à n dimensions). Exemple:

```
liste1 = ['toto', 'tata']
liste2 = ['titi', 'tutu', liste1]
print liste2[2][0]                --> toto
```

Manipulation de variables (suite)

Les Dictionnaires

Le dictionnaire est un système très souple pour intégrer des données. Si on pense aux listes comme une collection d'objets ordonnés et portant un indice par rapport à la position, un dictionnaire est une liste comportant à la place de ces indices, un mot servant de clé pour retrouver l'objet voulu. Un dictionnaire est affecté par une paire d'accolades {}.

Exemple:

```
dico = {'japon' :'japan', 'chine' :'china'}
print dico['chine']                    => china
```

Attention l'utilisation des dictionnaires (comme lors de l'affichage) se fait comme les listes: avec des []

Les clés peuvent être de toutes les sortes d'objet non modifiables. Les valeurs stockées peuvent être de n'importe quel type.

Un dictionnaire n'est pas ordonné comme une liste, mais c'est une représentation plus symbolique de classement par clés. La taille d'un dictionnaire peut varier, et le dictionnaire est un type modifiable sur place.

(dictionnaire = table de hash)

Manipulation de variables (suite)

Opération	Interprétation
d1 = {}	dictionnaire vide
d2={'one' : 1, 'two' : 2}	dictionnaire à duex éléments
d3={'count': {'one': 1, 'two': 2}}	inclusion
d2['one'], d3['count']['one']	indiçage par clé
d2.has_keys('one')	methodes : test d'appartenance
d2.keys()	liste des clés
d2.values()	liste des valeurs
len(d1)	longueur (nombre d'entrée)
d2[cle] = [nouveau]	ajout / modification
del d2[cle]	destruction

Manipulation de variables (suite)

Les tuples

Le tuple est comme la liste, une collection ordonnée d'objets. Un tuple peut contenir n'importe quelle sorte d'objet, mais la grande différence avec la liste est que le tuple n'est pas modifiable sur place. Un tuple se déclare avec des valeurs entre parenthèses et non entre crochets comme la liste.

tuple = (0, 1.4, «world»)

Les tuples peuvent être indicés pour la recherche d'un élément, une extraction est aussi possible. Ils supportent toutes sortes d'objets et même un tuple dans un tuple. Mais les tuples ne sont pas modifiables, donc comme les chaînes, ils ne peuvent pas grandir ou diminuer de taille.

Manipulation de variables (suite)

Opération	Interprétation
n1= ()	un tuple vide
n1 = (0,)	un tuple à un élément (et non une expression)
n2 = (0,1,2,3)	un tuple à quatre éléments
n2 = 0,1,2,3	un autre tuple à quatre éléments
n3 = ('abc', ('def', 'ghi'))	tuple avec inclusion
t[i], n3[i][j]	indiçage
n1[i:j]	tranche
len(n1)	longueur
n1+n2	concaténation
n2 * 3	répétition
for x in n2	itération
3 in s2	test d'appartenance

Manipulation de variables (suite)

Les variables globales

Lors de la création d'une variable dans une fonction celle-ci n'est pas forcément visible depuis le module de base. C'est une sorte de variable "visible au plus haut niveau du module", une "déclaration" global doit se faire dans une fonction pour que la variable puisse être utilisée depuis un autre bloc.

```
def ma_fonc():
    global x            #Valeur global
    y = 12              #Valeur local
    return x*y

x = 'toto'
print ma_fonc()
```

Les exceptions

Il est possible d'écrire des programmes qui prennent en charge des exceptions.
Il ne s'agit pas de gérer un problème de syntaxe mais de contourner une incapacité du programme.

Exemple:

```
try:
    x = int(raw_input("Veuillez entrer un nombre: "))

except :
    print "Aille! Ce n'était pas un nombre valide.  Essayez encore..."
```

Les exceptions ne sont pas des « conditions » (if, else,...)

Les itérations (=boucles)

Boucles while

L'instruction Python "while" est la construction d'itération la plus générale. "while" exécute de manière répétitive le bloc indenté tant que le test de condition est réalisé. Si la condition est d'emblée fausse le bloc ne sera jamais exécuté.

"while" consiste en une ligne d'en-tête avec une expression de test, suivie d'un bloc de plusieurs instructions. Il peut y avoir une partie "else" optionnelle, qui est exécutée si le contrôle sort de la boucle sans utilisation de l'instruction break.

```
while <test> :
    <instructions>
else :
    <instructions>
```

Les itérations (=boucles)

Boucle for

La boucle "for" est la séquence d'itération en Python, elle permet de traverser les éléments de tout objets qui répond aux opérations d'indiçage de séquence.
"for" fonctionne sur les chaînes, les listes, les tuples et d'autres objets issus de classes.
La boucle "for" commence par une ligne d'en-tête qui spécifie une cible d'affectation, ainsi qu'un objet qui sera itérer.

```
for <cible> in <objet> :
    <instructions>
```

exemple:

```
a = ['chat', 'chien', 'cochon']
for x in a:
        print x, ('nombre caracteres :', len(x), ')'
```

=>

```
chat (nombre caracteres : 4)
chien (nombre caracteres : 5)
cochon (nombre caracteres : 6)
```

La boucle "for" en Python fonctionne différemment qu'en C, elle affecte les objets de l'élément séquence à la cible un par un. A chaque affectation elle exécute le corps de la boucle. "for" quitte la boucle une fois que tous les éléments de l'élément séquence ont été parcourus.
"for" génère automatiquement l'indiçage de séquence.

Les itérations (=boucles)

break, continue, pass et le else de boucle

Ces trois instructions, "pass", "break", et "continue", servent à gérer la continuité d'une boucle suivant les actions qui se passent à l'intérieur.

- L'instruction "**break**" a pour but de sortir de la boucle instantanément et de passer à la suite.
- "**continue**" saute au début de la boucle la plus imbriquée.
- "**pass**" ne fait rien du tout, mais comme on ne peut avoir une expression qui n'est pas suivie, "pass" peut servir à combler ce vide.
- La boucle "**else**" exécute le bloc si la boucle se termine normalement. L'instruction "break" annule le passage dans la boucle "else".

```
for <cible> in <objet> :
        if <test1>: break
        elif <test2>: continue
        else: <instructions>
else :
   <instructions>
```

Les itérations (=boucles)

Boucles à compteurs et intervalles

La boucle "for" englobe la plupart des boucles à compteurs et c'est donc le premier outil pour traverser une séquence. Avec les boucles à compteurs d'intervalles il est possible de créer des séquences afin de pouvoir spécialiser l'indiçage de "for"

range()

L'instruction range retourne une liste d'entiers croissants successifs qui peuvent être utilisés comme index.

"Range" peut avoir de un à trois arguments. S'il n'y a qu'un argument il représente le nombre d'éléments de la liste partant depuis zéro et indicé de un. Si "range" est appelé avec deux arguments, il s'agit de la borne de départ de la liste et celle d'arrivée. Chaque élément de la liste sera une valeur incrémentée de un, entre la borne de départ et celle d'arrivée (non comprise). Quand "range" comportent trois arguments il s'agit de la borne de départ de la liste, celle d'arrivée et le pas d'incrémentation. (**xrange()** idem range mais avec des tuples)

```
x = range(10)        x -->   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
y = range(10,20)     y -->   [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
z = range(10,20,3)   z -->   [10, 13, 16, 19]
```

Les itérations (=boucles)

Utilisation de « for », « range() » et « len() »

Pour parcourir les indices d'une séquence, combinez range() et len() comme ci-dessous:

```
a = ['Marie', 'avait', 'un', 'petit', 'mouton']
max = len(a)
for i in range(max):
        print i, a[i]
```

résultat:

```
0 Marie
1 avait
2 un
3 petit
4 mouton
```

Les fichiers

La commande open permet à l'intérieur d'un programme python d'accéder à un fichier.
mon_fichier = open(«fichier», «w»)

Les commandes associées au traitement des fichiers ne se résument que par des méthodes.
Le tableau suivant montre les principales méthodes en vigueur sur les fichiers, il est important de dire que, lors de l'ouverture de l'objet-fichier celui-ci prend la forme de chaîne dans le programme Python.

Les fichiers

Opération	Interprétation
sortie = open('/tmp/spam', 'w')	crée un fichier de sortie ('w' => écriture)
entre = open('donnee', 'r')	ouvre un fichier en entrée ('r' => lecture)
s = entre.read()	lit le fichier entier dans une chaîne
s = entre.read(N)	lit N octets (1 ou plus)
s = entre.readline()	lit la ligne suivante
L = entre.readlines()	lit la fichier dans une liste de lignes
sortie.write(s)	écrit s dans le fichier
sortie.writelines(L)	écrit toutes les lignes contenues pas L
sortie.close()	fermeture manuelle

Fonction « open »:

- 'r' pour ouvrir le fichier en lecture seule ('r' est facultatif et sera pris par défaut s'il est omis)
- 'w' pour ouvrir le fichier en écriture
- 'a' ouvre le fichier en ajout; les données écrites dans le fichier seront automatiquement ajoutées à la fin.
- 'r+' ouvre le fichier pour la lecture et l'écriture.

La méthode « close » ferme l'objet fichier en cours. Python, pour gérer l'espace mémoire, ferme dans certains cas le fichier lui-même, lorsqu'il n'est plus référencé nulle part. Cependant fermer un fichier manuellement permet une certaine clarté, ce qui est important dans une grande application.

Les fichiers

Exercice:

- créer un liste contenant 5 valeurs: "ligne1", "ligne2", ..., "ligne5"

- parcourir la liste avec un « for » pour modifier chaque valeur en y ajoutant le caractères "\n" (=retour à la ligne)

- afficher tous les éléments de la liste (avec un « for »)

Le résultat doit ressembler à ça:

ligne1

ligne2

ligne3

ligne4

ligne5

Les fichiers

Exercice (suite):

- créer un fichier 'fichier.txt' à l'emplacement actuel de votre script python ("./fichier.txt") (mode 'w')
- écrire dans ce fichier le contenu de la liste précédente
- fermer le fichier

note: méthodes à utiliser: open, writelines, close

On veut rajouter une ligne à la fin.

- Ouvrir le fichier et y écrire cette chaîne: "dernière ligne" sans effacer le contenu précédent (mode 'a')
- fermer le fichier

note: méthodes à utiliser: open, write, close

- afficher le contenu du fichier (ouverture en mode 'r')
- fermer le fichier

note: méthodes à utiliser: open, read, close

Les fichiers

Exercice (correction):

```
liste = ['ligne1', 'ligne2', 'ligne3', 'ligne4', 'ligne5']

for i in range(len(liste)):

    liste[i]=liste[i]+"\n"
    print liste[i]



monfichier = open('fichier.txt','w')
monfichier.writelines(liste)
monfichier.close()

monfichier = open('fichier.txt','a')
monfichier.write("derniere ligne")
monfichier.close()

monfichier = open('fichier.txt','r')
contenu = monfichier.read()
monfichier.close()

print contenu
```

Objets et POO

Définition

Objets et Programmation Orientée Objets (=POO) sont au centre de la manière Python fonctionne. Vous n'êtes pas obligé d'utiliser la POO dans vos programmes, mais comprendre le concept est essentiel pour devenir plus qu'un débutant. Entre autres raisons parce que vous aurez besoin d'utiliser les classes et objets fournis par la librairie standard.

Qu'est-ce qu'un Objet ?

Un objet est un concept. Le fait d'appeler des éléments de nos programmes objets est une métaphore - une manière utile de penser à propos d'eux. La manière de concevoir un programme en objet est assez différente de la manière avec des fonctions (programmation procédurale)

En Python les éléments de base de la programmation sont des choses comme des chaînes de caractères (strings), des dictionnaires, des entiers, des fonctions, etc.. Ils sont tous des objets. Cela signifie qu'ils peuvent avoir des choses en commun.

Programmation procédurale et POO

La programmation procédurale consiste à diviser votre programme en blocs réutilisables appelés procédures ou fonctions.

- Vous essayez autant que possible de garder votre code en blocs modulaires, en décidant de manière logique quel bloc est appelé.
- Cela demande moins d'effort pour visualiser ce que votre programme fait.
- Cela rend plus facile la maintenance de votre code
- Vous pouvez voir ce que fait une portion de code.
- Le fait d'améliorer une fonction (qui est réutilisée) peut améliorer la performance à plusieurs endroits dans votre programme.

La programmation procédurale maintient une séparation stricte entre votre code et vos données.

- Vous avez des variables, qui contiennent vos données, et des procédures. Vous passez vos variables à vos procédures qui agissent sur elles et peut-être les modifient.
- Si une fonction veut modifier ce que contient une variable en la passant à une autre fonction, elle a besoin d'accéder à la variable et à la fonction qu'elle appelle.
- Si vous faites des opérations compliquées, cela peut être avec beaucoup de variables et de fonctions.

Programmation procédurale et POO

Il se trouve que beaucoup d'opérations sont communes à des objets du même type. Par exemple beaucoup de langages incluent une manière de créer une version en minuscule d'une chaîne de caractères.

Il y a beaucoup d'opérations standards qui sont associées uniquement à des chaînes de caractères. Par exemple passer cette chaîne en minuscule, en majuscule, découper la chaîne, etc…
Dans un langage orienté objet, nous construisons ces opérations en tant que propriété de l'objet chaîne de caractères. En Python cela s'appele des méthodes.

Chaque objet a un jeu de méthodes standards, vous en avez déjà utilisé certaines lors de la manipulation des fichiers (fichier.open, fichier.readlines(), ...).

Objets et POO

Exemple de méthodes pour les chaînes:

```
original_string = ' un texte '

# enlever l'espace du début et de la fin
string1 = original_string.strip()

# passé en majuscule
string2 = string1.upper()
print string2                    --->  UN TEXTE
```

Donc chaque chaîne de caractères est en fait un objet chaîne de caractères et possède toutes les méthodes d'un objet de type chaîne de caractères.
Dans le modèle objet, les méthodes et autres attributs qui sont associés avec un type particulier d'objet deviennent une partie de l'objet. Les données et les méthodes (=fonctions) pour manipuler les données ne sont plus séparées, mais sont liées ensemble dans un seul objet.

Avantages de la POO

Les objets combinent les données et les méthodes manipulant ces données. Cela signifie qu'il est possible de manipuler des choses compliquées en présentant une interface simple.

La manière dont ces opérations sont exécutées à l'intérieur de l'objet devient juste un détail de l'implémentation. Toute personne utilisant l'objet a juste besoin de connaître les méthodes publiques et les attributs. Ceci est le principe de l'**encapsulation**. D'autres parties de votre application (ou même d'autres programmeurs) peuvent utiliser vos classes et leurs méthodes publiques et mais vous pouvez mettre à jour l'objet sans rendre inutilisable l'interface qu'ils utilisent.

Vous pouvez également passer des objets au lieu de juste passer des données. Cela est l'un des aspects les plus utiles de la programmation orientée objet. A partir du moment où vous avez une référence vers l'objet vous pouvez accéder n'importe quel attribut de l'objet.

Le principal avantage de l'objet est que c'est une image utile. Cela correspond à la manière dont nous pensons. Dans la vie les objets ont des propriétés et inter-agissent les uns avec les autres. Si notre langage de programmation correspond à notre manière de penser, il sera plus facile de l'utiliser pour penser de manière creative.

Objets et POO

Notions à retenir

•- Les 'fonctions' qui font partie de l'objet sont appelées **méthodes**.

•

•- Les valeurs de l'objet sont appelées **attributs**.

•Note: vous pouvez examiner toutes les méthodes et attributs associés à un objet avec la commande dir : exemple : print dir(obj)

•

- Un objet regroupe des variables et des méthodes

- L'état d'un objet correspond à l'état de ses variables

- L'objet défini ses comportements par le biais de méthodes

- Il y a **encapsulation** des variables dans l'objet.

Exemples d ’objets

- Un client est un objet caractérisé par un nom, un prénom, une date de naissance, une profession ...

- Un compte en banque est caractérisé par un numéro de compte, un propriétaire (considéré comme un client), un taux d ’intérêt, un solde, un solde minimum, … et offre des services : consultation du solde, retrait, virement …

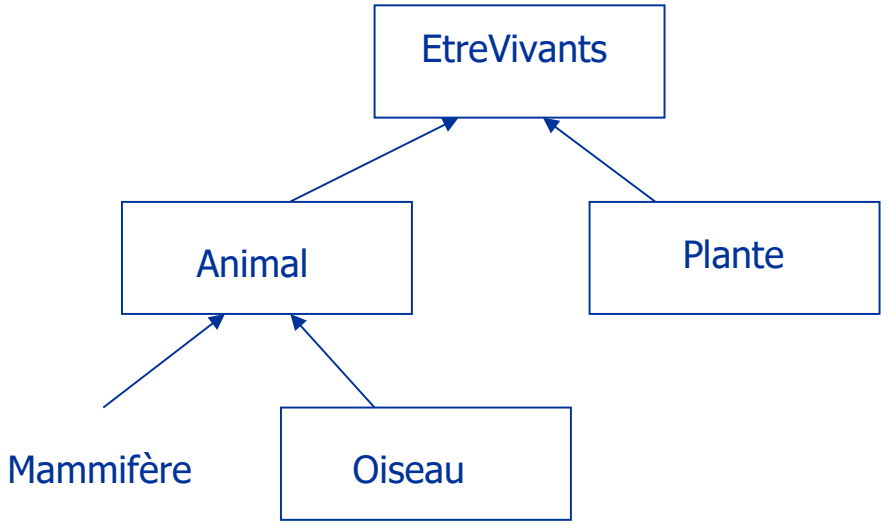
- Un objet graphique est caractérisé par une taille, par une position, une forme, une couleur … et peut changer de taille, de forme, de couleur ...

Le concept de classe

- Une classe est une “empreinte” ou un “moule” qui définit les variables et les méthodes communes à tous les objets de la classe.

- Ainsi, après avoir créé une classe, il faut instancier (créer une instance) avant de pouvoir l’utiliser. Un nouvel objet est alors créé.

- Une classe peut **hériter** d'une ou plusieurs autes classes. Cela signifie qu'elle partagera les attributs et les méthodes de ses parents



Création de classes

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction **class**. Nous allons donc apprendre à utiliser cette instruction, et pour commencer nous allons définir un type d'objet élémentaire, lequel sera simplement un nouveau type de donnée. Nous allons maintenant créer un nouveau type composite : le type Point.

Ce type correspondra au concept mathématique de point.
Dans un espace à deux dimensions, un point est caractérisé par deux nombres (ses coordonnées). En notation mathématique, on représente souvent un point par ses deux coordonnées x et y enfermées dans une paire de parenthèses. On parlera par exemple du point (25,17).
Une manière naturelle de représenter un point sous Python serait d'utiliser deux valeurs de type float. Nous voudrions cependant combiner ces deux valeurs dans une seule entité, ou un seul objet. Pour y arriver, nous allons définir une classe Point() :

Objets et POO

```
class Point:
    "Définition d'un point mathématique"
```

Les définitions de classes (=commentaires) peuvent être situées n'importe où dans un programme, mais on les placera en général au début (ou bien dans un module à importer).

N'oubliez pas le double point obligatoire à la fin de la ligne, et l'indentation du bloc d'instructions qui suit. Ce bloc doit contenir au moins une ligne. Dans notre exemple ultra-simplifié, cette ligne n'est rien d'autre qu'un simple commentaire. (Par convention, si la première ligne suivant l'instruction class est une chaîne de caractères, celle-ci pourra être incorporée automatiquement dans un dispositif de documentation des classes qui fait partie intégrante de Python. Prenez donc l'habitude de toujours placer une chaîne décrivant la classe à cet endroit. Faites de même pour les fonctions).

Une convention consiste à toujours donner aux classes des noms qui commencent par une majuscule.

Instance et variables d'instance

Nous venons de définir une classe Point(). Il nous reste maintenant à nous en servir pour créer des objets de ce type, par instanciation. Créons par exemple un nouvel objet 'monpoint' :

```
monpoint = Point()
```

Après cette instruction, la variable 'monpoint' contient la référence d'un nouvel objet Point(). Nous pouvons dire également que 'monpoint' est une nouvelle instance de la classe Point().

Nous pouvons ajouter des composants à un objet (une instance) en utilisant le système de qualification des noms par points :

```
monpoint.x = 3.0
monpoint.y = 4.0
```

Les variables ainsi définies sont des variables d'instance. Elles sont incorporées, ou plutôt encapsulées dans l'objet.

Passage d'objets comme arguments

Les fonctions peuvent utiliser des objets comme paramètres (elles peuvent également fournir un objet comme valeur de retour). Par exemple, vous pouvez définir une fonction telle que celle-ci :

```
def affiche_point(p):
    print "coord. horizontale =", p.x, "coord. verticale =", p.y
```

Le paramètre p utilisé par cette fonction doit être un objet de type Point(), puisque l'instruction qui suit utilise les variables d'instance p.x et p.y. Lorsqu'on appelle cette fonction, il faut donc lui fournir un objet de type Point() comme argument.

Exemple avec l'objet 'point' :

```
affiche_point(point)        -->        coord. horizontale = 3.0 coord. verticale = 4.0
```

Objets composés d'objets

Supposons maintenant que nous voulions définir une classe pour représenter des rectangles. Pour simplifier, nous allons considérer que ces rectangles seront toujours orientés horizontalement ou verticalement, et jamais en oblique.

De quelles informations avons-nous besoin pour définir de tels rectangles ?
Il existe plusieurs possibilités. Nous pourrions par exemple spécifier la position du centre du rectangle (deux coordonnées) et préciser sa taille (largeur et hauteur).
Nous pourrions aussi spécifier les positions du coin supérieur gauche et du coin inférieur droit. Ou encore la position du coin supérieur gauche et la taille. Admettons que ce soit cette dernière méthode qui soit retenue.

Objets et POO

```
class Rectangle:                                    <--   la nouvelle classe
    "définition d'une classe de rectangles"

boite = Rectangle()                                 <--   et une instance
boite.largeur = 50.0
boite.hauteur = 35.0
```

Nous créons ainsi un nouvel objet Rectangle() et deux variables d'instance. Pour spécifier le coin supérieur gauche, nous allons utiliser une instance de la classe Point() que nous avons définie précédemment. Ainsi nous allons créer un objet à l'intérieur d'un autre objet !

```
boite.coin = Point()
boite.coin.x = 12.0
boite.coin.y = 27.0
```

Pour accéder à un objet qui se trouve à l'intérieur d'un autre objet, on utilise la qualification des noms hiérarchisée (à l'aide de points) que nous avons déjà rencontrée à plusieurs reprises.

Création de méthodes

Pour aller un peu plus loin, nous allons définir encore une nouvelle classe, destinée cette fois à mémoriser des instants, des durées, etc. :

```
class Time:
  "Définition d'une classe temporelle"
```

Créons à présent un objet de ce type, et adjoignons-lui des variables d'instance pour mémoriser les heures, minutes et secondes :

```
instant = Time()
instant.heure = 11
instant.minute = 34
instant.seconde = 25
```

On crée la fonction affiche_heure(), qui sert à visualiser le contenu d'un objet de classe Time():

```
def affiche_heure(t):
    print str(t.heure) + ":" + str(t.minute) + ":" + str(t.seconde)
```

Objets et POO

Si par la suite vous utilisez fréquemment des objets de la classe Time(), il y est probable que cette fonction d'affichage vous sera fréquemment utile.

Il est alors judicieux d'**encapsuler** cette fonction dans la classe Time(),de manière à s'assurer qu'elle soit toujours automatiquement disponible chaque fois que l'on doit manipuler des objets de la classe Time().

Une fonction qui est ainsi encapsulée dans une classe s'appelle une **méthode**.

On définit une méthode comme on définit une fonction, avec cependant deux différences :

- La définition d'une méthode doit être placée à l'intérieur de la définition d'une classe, de manière à ce que la relation qui lie la méthode à la classe soit clairement établie.

- Le premier paramètre utilisé par une méthode est toujours le mot réservé "self ". Ce mot réservé désigne l'instance à laquelle la méthode sera associée, dans les instructions internes à la définition. (La définition d'une méthode comporte donc toujours au moins un paramètre, alors que la définition d'une fonction peut n'en comporter aucun).

Objets et POO

Pour réécrire la fonction affiche_heure() comme une méthode, nous allons donc déplacer sa définition à l'intérieur de celle de la classe, et changer le nom de son paramètre :

```
class Time:
    "Nouvelle classe temporelle"
    def affiche_heure(self):
        print str(self.heure) + ":" + str(self.minute) + ":" + str(self.seconde)
```

La définition de la méthode fait maintenant partie du bloc d'instructions indentées après l'instruction class. Notez bien l'utilisation du mot réservé self , qui se réfère donc à toute instance susceptible d'être créée à partir de cette classe.

Objets et POO

Nous pouvons à présent instancier un objet de cette classe :

```
maintenant = Time()
```

Si nous essayons d'utiliser un peu trop vite notre nouvelle méthode, ça ne marche pas :

```
maintenant.affiche_heure()          -->    AttributeError: 'Time' instance has no attribute 'heure'
```

C'est normal : nous n'avons pas encore créé les variables d'instance. Il faudrait faire par exemple :

```
maintenant.heure = 13
maintenant.minute = 34
maintenant.seconde = 21
maintenant.affiche_heure()                  --> 13:34:21
```

La méthode _init_ : valeurs par défaut pour les variables d'instance

L'erreur que nous avons rencontrée au paragraphe précédent n'est pas très plaisante. Il serait sans doute préférable que la méthode affiche_heure() puisse toujours afficher quelque chose, même si nous n'avons encore fait aucune manipulation sur l'objet nouvellement créé. En d'autres termes, il serait judicieux que les variables d'instance soient définies elles-aussi à l'intérieur de la classe, avec pour chacune d'elles une valeur "par défaut".

Pour obtenir cela, nous devons faire appel à une méthode particulière, que Python exécutera automatiquement lors de l'instanciation d'un objet à partir de sa classe. Une telle méthode est souvent appelée un constructeur. On peut y placer tout ce qui peut être nécessaire pour initialiser automatiquement l'objet que l'on crée.

Sous Python, une méthode est automatiquement reconnue comme un constructeur si on lui donne le nom réservé **__init__** (deux caractères "souligné", le mot init, puis encore deux caractères "souligné").

Objets et POO

Exemple :

```
class Time:
    "Encore une nouvelle classe temporelle"
    def __init__(self):
        self.heure =0
        self.minute =0
        self.seconde =0
    def affiche_heure(self):
        print str(self.heure) + ":" + str(self.minute) + ":" + str(self.seconde)

tstart = Time()
tstart.affiche_heure()          --> 0:0:0
```

L'intérêt de cette technique apparaîtra plus clairement si nous ajoutons encore quelque chose. Comme toute méthode qui se respecte, la méthode __init__() peut être dotée de paramètres. Ceux-ci vont jouer un rôle important, parce qu'ils vont permettre d'instancier un objet et d'initialiser certaines de ses variables d'instance en une seule opération. Dans l'exemple ci-dessus, veuillez donc modifier la définition de la méthode __init__() comme suit :

```
def __init__(self, hh =0, mm =0, ss =0):
    self.heure = hh
    self.minute = mm
    self.seconde = ss
```

Les arguments qui seront transmis à la méthode __init__() sont ceux que nous placerons dans les parenthèses qui accompagnent le nom de la classe, dans l'instruction d'instanciation.

Objets et POO

Voici par exemple la création et l'initialisation simultanées d'un nouvel objet Time() :

```
>>> recreation = Time(10, 15, 18)
>>> recreation.affiche_heure()                    -->   10:15:18
```

Puisque les variables d'instance possèdent maintenant des valeurs par défaut, nous pouvons aussi créer de tels objets Time() en omettant une partie des arguments :

```
>>> rentree = Time(10, 30)
>>> rentree.affiche_heure()              -->       10:30:0
```

Espaces de noms des classes et instances

Vous avez appris précédemment que les variables définies à l'intérieur d'une fonction sont des variables locales, inaccessibles aux instructions qui se trouvent à l'extérieur de la fonction. Cela vous permet d'utiliser les mêmes noms de variables dans différentes parties d'un programme, sans risque d'interférence.

Pour décrire la même chose en d'autres termes, nous pouvons dire que chaque fonction possède son propre espace de noms, indépendant de l'espace de noms principal.

Vous avez appris également que les instructions se trouvant à l'intérieur d'une fonction peuvent accéder aux variables définies au niveau principal, mais en lecture seulement : elles peuvent utiliser les valeurs de ces variables, mais pas les modifier (à moins de faire appel à l'instruction global).

Objets et POO

Il existe donc une sorte de hiérarchie entre les espaces de noms. C'est la même chose à propos des classes et des objets:

- Chaque classe possède son propre espace de noms. Les variables qui en font partie seront appelées les attributs de la classe.

- Chaque objet instance (créé à partir d'une classe) obtient son propre espace de noms. Les variables qui en font partie sont appelées variables d'instance ou attributs d'instance.

- Les classes peuvent utiliser (mais pas modifier) les variables définies au niveau principal .

- Les instances peuvent utiliser (mais pas modifier) les variables définies au niveau de la classe et les variables définies au niveau principal.

Objets et POO

Considérons par exemple la classe Time() définie précédemment.
Nous avons instancié deux objets de cette classe : recreation et rentree. Chacun a été initialisé avec des valeurs différentes, indépendantes. Nous pouvons modifier et réafficher ces valeurs à volonté dans chacun de ces deux objets sans que l'autre n'en soit affecté :

```
recreation.heure = 12

rentree.affiche_heure()          -->          10:30:0

recreation.affiche_heure()       -->          12:15:18
```

Objets et POO

Autre exemple :

```
class Espaces:                      # 1
    aa = 33                         # 2
    def affiche(self):              # 3
        print aa, Espaces.aa, self.aa   # 4

aa = 12                             # 5
essai = Espaces()                   # 6
essai.aa = 67                       # 7

essai.affiche()                     # 8      -->      12 33 67
print aa, Espaces.aa, essai.aa      # 9      -->      12 33 67
```

Dans cet exemple, le même nom aa est utilisé pour définir trois variables différentes : une dans l'espace de noms de la classe (à la ligne 2), une autre dans l'espace de noms principal (à la ligne 5), et enfin une dernière dans l'espace de nom de l'instance (à la ligne 7).
La ligne 4 et la ligne 9 montrent comment vous pouvez accéder à ces trois espaces de noms (de l'intérieur d'une classe, ou au niveau principal), en utilisant la qualification par points.

Héritage

Les classes constituent le principal outil de la programmation orientée objet, qui est considérée de nos jours comme la technique de programmation la plus performante. L'un des principaux atouts de ce type de programmation réside dans le fait que l'on peut toujours se servir d'une classe préexistante pour en créer une nouvelle qui possédera quelques fonctionnalités supplémentaires. Le procédé s'appelle dérivation. Il permet de créer toute une hiérarchie de classes allant du général au particulier.

Nous pouvons par exemple définir une classe Mammifere(), qui contiendra un ensemble de caractéristiques propres à ce type d'animal. A partir de cette classe, nous pourrons alors dériver une classe Primate(), une classe Rongeur(), une classe Carnivore(), etc., qui hériteront de toutes les caractéristiques de la classe Mammifere()

Objets et POO

Exemple :

```
class Mammifere:
    caract1 = "il allaite ses petits;"

class Carnivore(Mammifere):
   caract2 = "il se nourrit de la chair de ses proies;"

class Chien(Carnivore):
   caract3 = "son cri s'appelle aboiement;"

mirza = Chien()
print mirza.caract1, mirza.caract2, mirza.caract3   -->   il allaite ses petits; il se nourrit de la chair de ses proies; son cri s'appelle aboiement;
```

Dans cet exemple, nous voyons que l'objet mirza, qui est une instance de la classe Chien(), hérite non seulement de l'attribut défini pour cette classe, mais également des attributs définis pour les classes parentes.

Vous voyez également dans cet exemple comment il faut procéder pour dériver une classe à partir d'une classe parente : on utilise l'instruction class , suivie comme d'habitude du nom que l'on veut attribuer à la nouvelle classe, et ensuite on place entre parenthèses le nom de la classe parente.

Les classes

Exo

- Créer une classe Personne qui contient 3 attributs: nom, prenom et age avec comme valeurs par défauts "", "" et "25"

- Créer dans cette classe une methode afficheToi, sans paramètre (hormis le self...) qui affiche le prénom, le nom et l'age.

- Instancier un objet p en utilisant cette classe

- Assigner le variables d'instances (de l'objet p) nom et prénom avec les valeurs "Toto" et "Jean"

- Faites appel à la méthode afficheToi pour afficher les valeurs de cet objet

Les classes

Correction:

```
class Personne:

    nom=""
    prenom=""
    age=25

    def afficheToi(self):
        print self.prenom,' ',self.nom,' a ',self.age



p=Personne();
p.nom="Toto"
p.prenom='Jean'

p.afficheToi()
```

Bibliothèques graphiques

Une bibliothèque graphique est une bibliothèque logicielle spécialisée dans les fonctions graphiques. Elle permet d'ajouter des fonctions graphiques à un programme.

Ces fonctions sont classables en trois types qui sont apparus dans cet ordre chronologique et de complexité croissante :

- Les bibliothèques de tracé d'éléments 2D

- Les bibliothèques d'interface utilisateur

- Les bibliothèques 3D

Bibliothèques graphiques

Les bibliothèques de tracé d'éléments 2D

Ces bibliothèques sont également dites bas niveau. Elles permettent de tracer les éléments graphiques de base que sont les lignes, les polygones et d'afficher des pixels ce qui permet d'afficher des icônes et de images.

Les bibliothèques graphiques peuvent communiquer directement avec le matériel c'est à dire la mémoire vidéo ou une carte graphique ou bien utiliser un pilote.

La bibliothèque X Window System sous Unix est typiquement une bibliothèque dédiée principalement à ce type de fonctions.

Certains anciens langages comme le BASIC comprenaient des fonctions graphiques comme partie intégrante du langage.

Bibliothèques graphiques

Les bibliothèques d'interface utilisateur

Les interfaces utilisateurs sont les éléments graphiques qui permettent à l'utilisateur d'interagir avec le programme. Apparues avec l'ordinateur STAR de Xerox, elles sont maintenant la base de l'ergonomie des ordinateurs.

Elles permettent de construire une représentation graphique au programme avec des fenêtres, boutons, ascenseurs.

Dans les bibliothèques d'interface utilisateur on peut citer Motif, Qt, GTK, GNOME, Win32.

Bibliothèques graphiques

Les bibliothèques 3D

Apparues en dernier chronologiquement, les bibliothèques 3D permettent de faire de la synthèse d'image 3D c'est à dire de dessiner des éléments en volume.

La première bibliothèque 3D était faite par Silicon Graphics: GL devenue OpenGL par la suite est l'une des plus connues avec DirectX de Microsoft.

Les bibliothèques 3D actuelles font toutes appel à une carte accélératrice.

Bibliothèques graphiques

L'utilisation que nous ferons du langage Python se limitera au déjà vaste domaine des interfaces graphiques (ou GUI : Graphical User Interface en anglais et IHM : Interface Homme Machine en français).

Ce domaine est en effet extrêmement complexe : chaque système d'exploitation peut proposer plusieurs "librairies" de fonctions graphiques de base, auxquelles viendront fréquemment s'ajouter de nombreux compléments, plus ou moins spécifiques de langages de programmation particuliers. Tous ces composants sont généralement présentés comme des classes d'objets, dont il vous faudra étudier les propriétés et les méthodes.

Note: Bibliothèques = Libraries (en anglais) = Librairies (francisé)

La bibliothèque Tkinter

Tkinter est la bibliothèque graphique de base pour le langage Python. Elle est limitée mais simple d'utilisation.

A noter que vous pouver télécharger un package contenant Python, plusieurs bibliothèques et Tkinter, très facile d'installation : *http://www.activestate.com/activepython/downloads*

Tkinter vient d'une adaptation de la librairie graphique Tk écrite pour Tcl. Tk et Tkinter sont disponibles sur la plupart des plates-formes Unix (dont les tous les Linux), Windows et Macintosh.

Pour utiliser Tkinter, tout ce que vous devez faire (après avoir installé la bibliothèque en plus de Python) est importer cette biblothèque :

import Tkinter

ou alors :

from Tkinter import *

A noter que Tk est un toolkit graphique multi plate-formes qui prend l'apparence du système sur lequel on l'utilise (=les fenetres auront le look du systeme d'exploitation installé sur la machine)

Premier exemple - Le widget « Label »

from Tkinter import * # importer toutes les classes (*) contenues dans le module Tk

fen = Tk() # créer une fenêtre (instance de la classe Tk - "Instance" <=> ~ fonction en langage objet)

texte = Label(fen, text="Bienvenue !", fg='red') # création de l'objet texte (par instanciation de la widget "Label").
#Le 1er paramètre est le nom de la fenetre principal (ici nommé "fen"), ensuite le texte simple, et enfin des arguments optionnels comme ici la couleur du texte en rouge

texte.pack() **#** la méthode (methode <=> #fonction en langage objet) "pack()" affiche l'élément dans la fenêtre

bouton= Button(fen, text="Quit", command=fen.quit) **#** création de l'objet bouton (par instanciation de la classe Button), dans "fen"

bouton.pack() **#** placement de l'objet bouton

fen.mainloop() # fait une boucle sur la fenetre pour éviter qu'elle se referme automatiquement à la fin du programe. Plus précisement "mainloop" gère les événements (click souris, clavier) et le système de gestion de fenêtre (rafraîchissement). Cette ligne de commande devra donc être la dernière de tout programme

Présentation des widgets

Principaux widgets:

Les composants graphiques de Tkinter sont appelés widgets.

Il existe 15 classes de widgets dans le module Tkinter :

A chacun de ces widgets est associé un grand nombre d'options et de méthodes.

On peut aussi leur lier des événements

Widget	Description
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utiliser pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
Checkbutton	Une case à cocher qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état.
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
Frame	C'est le widget "contenant". Il permet de grouper plusieurs autres widgets. Il peut posséder une bordure ou un fond coloré.
Label	Un texte quelconque (éventuellement une image).
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la "Listbox" de telle manière qu'elle se comporte comme une série de "boutons radio" ou de cases à cocher.
Menu	Ce peut être un Menu déroulant attaché à la barre de titre, ou bien un Menu "popup" apparaissant à la suite d'un clic.
Message	Permet d'afficher un texte. Ce widget est une variante du widget Label. Il permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur.
Radiobutton	Représente la valeur exclusive d'une variable : parmi une liste de choix, une seule valeur est autorisée. Cliquer sur un "bouton radio" donne la valeur correspondant à la variable, et "vide" tous les autres boutons radio associés à la même variable.
Scale	Permet de fixer de manière très visuelle une valeur numérique, en déplaçant un curseur le long d'un axe.
Scrollbar	"ascenseur" ou "barre de défilement" utilisable en association avec les autres widgets : Canvas, Entry, Listbox, Text.
Text	Affichage de texte formaté. Permet aussi à l'utilisateur d'éditer le texte affiché. Des images peuvent également être insérées.
Toplevel	Un widget " contenant " affiché séparément, il prend la main.

Présentation des widgets

Options communes:

Option	Description	Tous les widgets sauf
Width	Valeur positive indiquant la largeur du widget en unité de caractère dans la police donnée par l'option font . Si négatif ou nul, le widget s'adapte automatiquement.	Menu
Height	Valeur positive indiquant la hauteur du widget en unité de caractère dans la police donnée par l'option font. Doit valoir au moins 1.	Entry, Menu, Message, Scale, Scrollbar
background (bg)	Couleur de l'arrière plan du widget	
foreground (fg)	Couleur du premier plan du widget.	Canvas, Frame, Scrollbar, Toplevel
Relief	Effet 3D du widget ; valeurs possibles : RAISED, SUNKEN, FLAT, RIDGE, SOLID et GROOVE. Cette valeur indique comment l'intérieur du widget apparaît par rapport à son extérieur.	
borderwidth (bd)	Valeur positive indiquant la largeur de la bordure 3D à dessiner à l'extérieur du widget (si une telle bordure est dessinée ; déterminé par l'option relief)	
Takefocus	Booléen indiquant si le widget accepte un focus clavier. <Shift – Tab>	
highlightcolor	Couleur du rectangle qui apparaît autour du widget lorsque le focus clavier est sur celui-ci.	Menu
highlightbackground	Couleur de la région autour du widget lorsqu'il n'a pas le focus clavier.	Menu
highlightthickness	Valeur positive indiquant la largeur des bords du rectangle du focus clavier.	Menu
Font	Police pour écrire du texte à l'intérieur du widget	Canvas, Frame, Scrollbar, Toplevel
Cursor	Curseur de la souris à utiliser dans le widget.	

Présentation des widgets

Il est possible de lire/modifier ces options, à l'aide de trois méthodes :

get(option) : retourne la valeur courante d'une option sous forme d'une chaîne de caractère.

configure(option=valeur,…) , config(option=valeur,…) : modifie une ou plusieurs options.

keys() : retourne la liste de toutes les options qui peuvent être modifiées.

Présentation des widgets

Gestion du positionement des objets graphiques

Gestionnaire	Description
Grid	Le gestionnaire d'agencement crée des affichages style tableau, en organisant les widgets dans une grille 2D.
Pack	Le gestionnaire d'agencement pack permet de positionner (agencer) les widgets en les empaquetant dans un widget parent. Les widgets sont traités comme des blocs rectangulaires placés dans un cadre.
Place	Le gestionnaire d'agencement place explicitement un widget dans une position donnée.

Le widget « Label »

Un widget Label peut afficher du texte, une icône ou une image (pour afficher plusieurs lignes de texte, le widget Message est plus adapté car il offre plus d'options de mise en forme)

Options :

Option	Type	Description
text	string	Texte affiché dans le libellé, sur une ou plusieurs lignes. Si les options "bitmap" ou "image" sont spécifiées, cette option est ignorée.
bitmap	bitmap	"Bitmap" à afficher dans le widget Label. Si l'option "image" est spécifiée, cette option est ignorée. Les bitmaps suivants sont toujours disponibles : error, gray75, gray50, gray25, gray12, hourglass, info, questhead, question, warning. On peut également charger un "bitmap" à partir d'un fichier XBM, par exemple, "@sample.xbm".
image	image	Image affichée dans le widget Label. Si cette option est précisée, elle prend le dessus sur les options "text" et "bitmap".
justify	constant	Définit la façon d'aligner plusieurs lignes de texte (LEFT, RIGHT ou CENTER).
anchor	constant	Situation du texte ou de l'image dans le libellé. Utilisez N, NE, E, SE, S, SW, W, NW, ou CENTER (par défaut).
wraplength	distance	Détermine quand le texte du libellé doit être coupé. Cette option est donnée en unités d'écran. Par défaut, le texte est affiché sur une seule ligne.
textvariable	variable	Associe une variable Tkinter (généralement StringVar) au libellé. Si la variable est modifiée, le texte du libellé est mis à jour.
underline	int	Précise quel caractère est souligné dans le texte. -1 (par défaut) signifie qu'aucun n'est souligné.

Le widget «Button»

Ce widget permet à l'utilisateur de dire " Fais ceci maintenant ", où " ceci " est décrit par le texte sur le bouton ou suggéré par son icône. Les boutons sont souvent utilisés pour accepter ou refuser des données entrées dans une boite de dialogue.

Méthodes :

flash( ) : fait clignoter le bouton entre l'état normal et actif.

invoke( ) : simule un appel de la commande associée au bouton.

Options :

Option	Type	Description
anchor	constant	Position du texte ou de l'image dans le bouton. Utilisez N, NE, E, SE, S, SW, W, NW, ou CENTER (par défaut). Si cette option est modifiée, il serait souhaitable d'utiliser les options padx et pady
bitmap	bitmap	"Bitmap" à afficher dans le widget bouton. Si l'option "image" est spécifiée, cette option est ignorée. Les bitmaps suivants sont toujours disponibles : error, gray75, gray50, gray25, gray12, hourglass, info, questhead, question, warning.
command	callback	Fonction ou méthode appelée quand le bouton est appuyé.
disabledforeground	color	Couleur du texte ou de l'image quand le bouton est inaccessible. La couleur de fond est inchangée.
image	image	Image affichée dans le widget bouton. Si cette option est précisée, elle prend le dessus sur les options "text" et "bitmap".
justify	constant	Définit la façon d'aligner plusieurs lignes de texte (LEFT, RIGHT ou CENTER).
padx, pady	distance	Marges horizontale et verticale entre le texte ou l'image et le cadre du bouton.
state	constant	Etat du bouton : NORMAL (par défaut), ACTIVE ou DISABLED.
text	string	Texte affiché dans le bouton. Si les options "bitmap" ou "image" sont précisées, cette option est ignorée.
textvariable	variable	Associe une variable Tkinter (généralement StringVar) au bouton. Si la variable est modifiée, le texte du bouton est mis à jour.
underline	int	Précise quel caractère est souligné dans le texte. -1 (par défaut) signifie qu'aucun n'est souligné.
wraplength	distance	Détermine quand le texte du bouton doit être coupé. Cette option est donnée en unités d'écran. Par défaut, il est affiché sur une seule ligne.

Le widget «Button»

Exemple: Fait apparaître un message à la suite d'un click de bouton.

```
from Tkinter import *

def affiche_info():

    lab=Label(root, text="avant dernier cours d'info. :-( ")
    lab.pack()

root=Tk()

but=Button(root, text="Info", command=affiche_info)

but.pack()

root.mainloop()
```

Le widget «Text»

Le widget Text est utilisé pour afficher des documents texte simples ou formatés (diverses polices de caractères et mises en forme sont disponibles).

Index :

Le widget Text vous permet de spécifier la position d'un caractère ou d'une zone de texte de plusieurs manières. Ligne colonne, Fin de ligne, INSERT, CURRENT, END, Etiquette, Sélection, Coordonnées souris, etc...

Méthodes (ces méthodes permettent d'utiliser les index):

- insert (index,text), insert(index,text,tags) : insère du texte à l'endroit donné (index).
- delete(index), delete(start,stop) : supprime le caractère de position "index" ou l'ensemble du texte situé entre start et stop.
- get(index), get(start,stop) : retourne le caractère de position "index" ou l'ensemble du texte situé entre start et stop.
- dump(index,options...), dump(start,stop,options...) : retourne la liste des composants du widget de position "index" donnée ou de l'ensemble du texte compris entre start et stop.
- index(index) : retourne l'indice "ligne/colonne" correspondant à l'indice donnée.
- ...

Liste des options et méthodes: sur **http://www.pythonware.com/library/tkinter/introduction/**

Le widget «Text»

```
from Tkinter import *

def aff_info():
        txt.insert(END,"Bienvenue dans le tutoriel de Tkinter\n")

root=Tk()

but=Button(root, text="Info", command=aff_info)
but.pack()

txt=Text(root)
txt.pack()

root.mainloop()
```

Le widget Text est créé et " packé " dans la fenêtre root :

La fonction aff_info() insère la chaîne de caractères dans la zone de texte. Elle utilise la méthode insert() du widget text. L'option END spécifie le point d'insertion

Le widget «Entry»

Le widget Entry est utilisé pour lire des chaînes de caractère. Il donne la possibilité à l'utilisateur d'entrer une ligne de texte avec une seule police de caractère. Pour entrer plusieurs lignes de texte, utiliser le widget Text

Méthodes :

- insert (index,text) : insère du texte au niveau de l'index spécifié. Utilisez insert(INSERT,text) pour insérer du texte au niveau du curseur et insert(END,text) pour ajouter en fin de texte
- delete (index), delete (from,to) : supprime le caractère situé à la position de l'index indiquée ou à l'intérieur d'un domaine donné. Utilisez delete(0,END) pour effacer l'intégralité du texte du widget.
- icursor (index) : déplace le curseur à la position de l'index indiquée. Ceci permet également de modifier l'index INSERT dont nous avons parlé ci-dessus.
- get ( ) : récupère le contenu du champ de saisie
- index (index) : retourne la position numérique de l'index spécifié.

Le widget «Entry»

Le widget entry est un widget d'entrée de texte simple.

```
from Tkinter import *
root=Tk()

saisie=StringVar()

lb1= Label(root, textvariable=saisie, width=30, foreground="blue")
ent=Entry(textvariable=saisie, width=30)
ent.pack()
lb1.pack()

root.mainloop()
```

Placement des widgets

Le packer

La commande pack permet de placer des widgets soit horizontalement (side=left ou rigth), soit verticalement (side=top).
Lorsqu'on veut passer d'un placement horizontal à un placement vertical on utilise des frames.

Placement des widgets

```
from Tkinter import *

root = Tk()

root.title("pack")

f=Frame(root)
msg="0"
lb1=Label(f, text="message")
entre=Entry(f, textvariable=msg)

lb2=Label(root, text="resultat:")
res=Label(root, textvariable=msg, fg="blue")

lb1.pack(side="left")          # On place les premiers widgets horizontalement dans la frame
entre.pack()

f.pack(side="top")                  # puis on place la frame et les autres widgets verticalement
lb2.pack()
res.pack()

root.mainloop()
```

Placement des widgets

Le grid

Le même exemple que précédemment, avec la commande grid:

```
from Tkinter import *
root = Tk()

msg="0"
lb1=Label(root,text="message")
entre=Entry(root,textvariable=msg)

lb2=Label(root,text="resultat:")
res=Label(root,textvariable=msg,fg="blue")

lb1.grid(row=0,column=0)
entre.grid(row=0,column=1)
lb2.grid(column=1)
res.grid(column=1)

root.mainloop()
```

Les options column et row permettent de spécifier le positionnement d'un objet.

Placement des widgets

Le placer

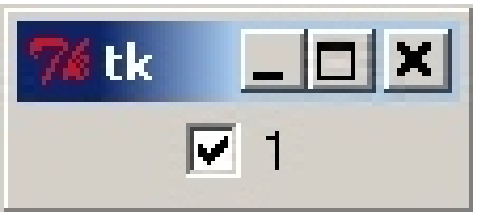
La commande place permet de placer un objet graphique (widget) à la position x y:

```
from Tkinter import *
root = Tk()
lb=Label(root,text="un label")
lb.place(x=50, y=50 )

root.mainloop()
```

Le widget «CheckButton»

Ce sont des cases à cocher.

Exemple:

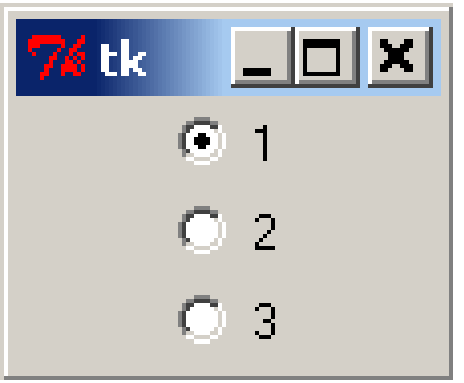



```
from   Tkinter import *
root=Tk()
val=1
chk=Checkbutton(root, textvariable=val, variable=val)
chk.grid()
root.mainloop()
```

Le widget «RadioButton»

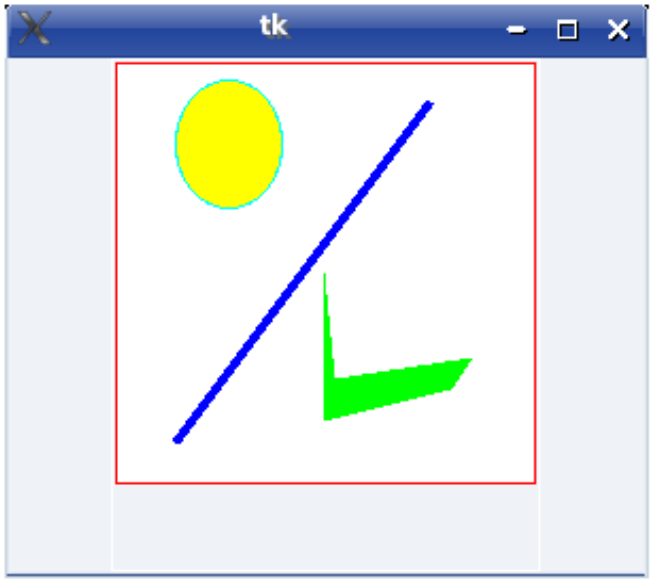
Un radiobutton ressemble un peu a un checkbutton, à la différence près que l'on peut associer plusieurs radiobutton à une même variable et définir pour chacun d'entre eux la valeur qui sera affectée à la variable lorsque celui-ci sera cochée.
Exemple:

```
from   Tkinter import *

root=Tk()

r1=Radiobutton(root, text="1", variable=myvar, value="1")
r2=Radiobutton(root, text="2", variable=myvar, value="2")
r3=Radiobutton(root, text="3", variable=myvar, value="3")

r1.grid()
r2.grid()
r3.grid()

root.mainloop()
```



Le widget «Canvas»

Un canvas est une zone dans laquelle on peut dessiner des lignes, rectangles , cercles, arcs, courbes ou encore afficher des images.



Exemple:

```
from Tkinter import *
root = Tk()

canv=Canvas(root,width=200,height=300)

canv.create_rectangle((2,2,199,199),fill="white",outline="red")
canv.create_line((150,20,30,180),fill="blue",width=4)
canv.create_oval((30,10,80,70),fill="yellow",outline="cyan")
canv.create_polygon((100,100, 105,150, 170,140, 160,155, 100,170),fill="green")

canv.pack()

root.mainloop()
```

Exercice

Générateur de rectangles:

Faire un programme qui trace des rectangles de tailles différentes en fonction de coordonnées entrées par un utilisateur:

- Faire un canvas gris qui servira de fond d'écran (de 400 par 400 par exemple)
- Faire 4 cases Entry correspondantes aux 4 coordonnées pour dessiner le rectangle (coin supérieur gauche et coin inférieur droite: x1, y1, x2, y2)
- Mettre 4 labels pour ces 4 cases (noms des cases par exemple)
- Faire un bouton pour quitter le programme et un autre pour tracer le rectangle
- Ce dernier bouton lancera une fonction qui dessinera le rectangle (méthode « create_rectangle » a appliquer sur le canvas de fond. Les coordonnées à passer en arguments sont les valeurs des 4 cases d'entrées (qu'il faudra récupérer avec la méthode get() exemple x = mon_objet.get())

Note: fill="une couleur" et outline="une autre couleur" sont des arguments que l'on peut utiliser pour un canvas (rectangle ou pas)

Correction

```
from Tkinter import *

def trace_rectangle ():                    # fonction qui dessine des rectangles
        x1=ent1.get()                              # récupérer les valeurs des boites d'entrées
        y1=ent2.get()
        x2=ent3.get()
        y2=ent4.get()
        canvas.create_rectangle((x1,y1,x2,y2), fill="blue", outline="red")

fen = Tk()                                         # creer une fenetre, instance de la classe Tk


canvas =  Canvas(fen, height=400,width=400,bg='dark grey')          # creer un canvas qui servira de fond
canvas.pack(side=TOP)

# creer les boites entry pour y mettre les valeurs ainsi que les labels

lab1 = Label(fen, text="x1")
lab1.pack(side=LEFT)
ent1 = Entry(fen, width=5)
ent1.pack(side=LEFT)
```

Correction (suite)

```
lab2 = Label(fen, text="y1")
lab2.pack(side=LEFT)
ent2 = Entry(fen, width=5)
ent2.pack(side=LEFT)

lab3 = Label(fen, text="x2")
lab3.pack(side=LEFT)
ent3 = Entry(fen, width=5)
ent3.pack(side=LEFT)

lab4 = Label(fen, text="y2")
lab4.pack(side=LEFT)
ent4 = Entry(fen, width=5)
ent4.pack(side=LEFT)

bouton1= Button(fen, text="Trace mon rectangle", command=trace_rectangle)          # boutons pour tracer
bouton1.pack()

bouton2 = Button(fen, text="Quitter", command=fen.quit)                            # bouton pour quitter
bouton2.pack(side=BOTTOM)

fen.mainloop()                                                                     # gestionnaire d evenement
```