Cours programmation - Marc Jachym
Licence professionnelle – Métrologie dimensionnelle et qualité
IUT de St Denis, Université Paris 13

Cours de Programmation avec le langage Python
Niveau débutant en programmation

Table des matières

Partie II– Les éléments de base du langage Python

II.1. Pourquoi Python ?
Quelques notions essentielles sur le langage Python

Python est un langage de programmation généraliste, facile à apprendre et rapide à mettre en œuvre.

Python est généraliste car, selon les multiples réalisations qu'il a à son actif, il peut être utilisé dans tous les domaines : écriture d'applications pour le Web (serveur d'application Zope, framework Django), programmes de calculs mathématiques (biblithèque SciPy), interfaces graphiques (il existe des supports de Python pour les sytèmes d'interface graphique GTK, Qt, TK, wxWidget), programmation de scripts systèmes, etc.

De fait, Python dispose d'une très large bibliothèque standard qui offre au programmeur des outils très divers pour : la gestion réseau (librairie socket), la manipulation du format xml, l'accès aux protocoles d'Internet (protocoles des services courriel, divers protocoles web), l'accès aux éléments du système d'exploitation sous-jacent (accès aux fichiers et répertoires, gestion des processus), l'écriture d'interfaces graphiques (librairie Tkinter), l'accès aux bases de données relationnelles, etc.
Il est aussi possible d'étendre Python en intégrant de nouveaux modules. Par exemple la librairie PIL permet de traiter des images.

[The Python Standard Library: http://docs.python.org/2/library/index.html]

Python est facile à apprendre car de nombreuses opérations dévolues au programmeur dans les langages classiques comme le langage C, par exemple la gestion de la mémoire, sont prises en charge par l'interpréteur Python. De même, Python gère dynamiquement les variables et libère le programmeur des déclarations de type.
De plus Python impose d'écrire les blocs d'instructions de manière indentée, ce qui favorise grandement la lecture des programmes.

Enfin, en tant que langage interprété (voir dans la partie I de ce cours), Python est rapide à mettre en œuvre. Il suffit de lancer la console Python pour avoir sous la min de quoi tester directement des commandes et des structures de données. Cela est un gain de temps pour le programmeur par rapport au cycle compilation/édition de liens du langage C.

Ceci étant dit, et s'il est vrai que Python peut être utilisé dans de nombreux domaines, cela ne signifie par que Python est le bon choix dans tous les cas !
Python est souvent considéré comme un langage de scripts, c'est à dire destiné à l'écriture de petits programmes utilitaires servant dans des contextes particuliers.
En tant que langage dynamique interprété, Python doit réaliser un nombre d'opérations bien plus grand que les lignes écrite par le programmeur dans son programme. En effet, il faut que Python garde en mémoire et gère des descripteurs pour chacun des symboles utilisés dynamiquement et qu'il gère les réservations et libérations de mémoire correspondantes.

L'exécution d'un programme Python sera donc toujours plus lente que celle d'un programme compilé. Ainsi un gros logiciel diffusé à grande échelle (prenons l'exemple d'un logiciel de traitement de texte ou d'un gestionnaire de bases de données) sera plus probablement écrit en langage C ou en C++.

Mais pour de nombreux usages, quand la rapidité d'exécution n'est pas le critère déterminant, Python est un bon choix.

Pour l'apprentissage de la programmation, c'est un bon choix également car cela permet une entrée en matière plus rapide et simplifiée. Il faut cependant avoir conscience que les facilités offertes par Python ou autres langages dynamiques cachent de nombreux détails au programmeur et pourraient donner une impression erronée au débutant en programmation. En effet la gestion mémoire d'un tableau de nombres entiers, par exemple, n'est pas une chose « allant de soi » et le programmeur en assembleur ou en C (ou dans le cas de Python : l'interpréteur) doit contrôler ce qui se passe en mémoire. Cela est en quelque sorte plus pédagogique.

Last but not least : Python est un langage **orienté objet** et nous verrons ce que cela signifie dans la partie III de ce cours.

II.2. Premiers éléments de programmation en Python

II.2.1 Les variables

Rappelons qu'une variable représente un stockage en mémoire vive de l'ordinateur (mémoire vive : là ou s'exécutent les programmes, par opposition à mémoire de masse, comme par exemple un disque dur où sont stockées des informations à long terme).

Ce **stockage** est manipulé grâce à un **nom symbolique**.
Contrairement à ce qui est habituel dans les langages dits de « bas niveau », où l'architecture matérielle de l'ordinateur est apparente au programmeur, le programmeur Python ne travaille jamais par accès explicite à la mémoire en manipulant des adresses (ni même des « pointeurs » comme en langage C). Tout

accès à la mémoire est réalisé par l'intermédiaire de variables qui ont une signification précise pour le programmeur.

Les noms des variables obéissent à des règles :
Un nom doit débuter par une lettre ou par le caractère de soulignement _ puis il doit être suivi par un nombre quelconque de lettres, chiffres ou de caractères de soulignement. On remarquera donc que le soulignement est le seul caractère « exotique » autorisé. Par exemple, le tiret n'est pas autorisé et un nom comme *ma-variable* n'est pas valable comme nom de variable.

Ex de noms valables : compteur, Unnombre, unNombre, un_nombre, _compteur1, __compteur12, u234, a, i2

Ex de noms invalides : 1nombre, ma-variable, code%secret,

Intéressant : On en profitera pour découvrir dans la documentation officielle de Python, la partie *The Python langage reference*. La partie 2, *Lexical analysis*, utilise la notation appelée *BNF grammar* pour présenter de manière complète et normalisée la syntaxe du langage Python.
Pour la description du nom des variables, voir *Identifiers ans Keywords* :
http://docs.python.org/2.7/reference/lexical_analysis.html#identifiers

La **création d'une variable** se fait lors de sa première utilisation qui est toujours une initialisation réalisée grâce à l'**opérateur d'affectation** =

Ex :

a = 4

Cette affectation créé une variable nommée **a** et lui affecte la valeur entière 4.
On peut dès lors utiliser cette variable.

Pour afficher son contenu :
print a

Pour la placer dans une autre variable :
b = a

Pour l'utiliser dans une **expression** :
c = 10 + 2*a

Rq importante :
Le signe = utilisé dans les exemples précédents est bien l'**opérateur d'affectation**. Il s'agit d'une **action** visant à placer dans le membre de gauche (c'est à dire le stockage

mémoire représenté par la variable placée à gauche du signe =) le contenu représenté par l'expression simple ou complexe placée à droite de l'opérateur.

Il ne faut surtout pas confondre cet opérateur avec le signe = des mathématiques. Ce dernier sert à affirmer une égalité. L'opérateur d'affectation, lui, n'a pas valeur d'affirmation mathématique mais représente une action.

II.2.2 Le type des variables

C'est une notion très importante en programmation. En effet, aux **valeurs** contenues en mémoire sont associés des **types**. Ces types permettent à la machine de savoir ce qu'elle peut faire avec les valeurs et comment les manipuler.

Par exemple : le caractère 'A' est codé en mémoire par la valeur 65, le caractère 'B' par la valeur 66.
Si les valeurs sont considérées comme des entiers, l'opérateur '+' représente l'addition :
65+66 a pour résultat 131

Si les valeurs sont considérées comme des (chaines de) caractères, l'opérateur '+' représente la concaténation (mise bout à bout) : 'A'+'B' a pour résultat 'AB'

En Python, contrairement à ce qui se passe pour d'autres langages comme le langage C, il n'y a **pas de déclaration de type lors de la création d'une variable** !

Par exemple, en langage C, la déclaration d'une variable de type entier, avec initialisation à une certaine valeur, se fait par la déclaration suivante :

```
int var = 16 ;
```

cela créé une variable de type entier nommée var.

En Python, cela donne :

```
var = 16
```

Les variables Python sont créées lors de leur première affectation, sans déclaration préalable.
Les variables Python ne sont donc pas associées à des types, mais les valeurs qu'elles contiennent, elles, le sont !
C'est au programmeur de savoir le type de la valeur contenue dans une variable à un moment donné pour ne pas faire d'opération incompatible.

Par exemple, voici deux instructions successives incompatibles tapées dans l'interpréteur Python :

```
>>> var = 16
>>> var = var + 'c'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Dans la seconde instruction, nous tentons d'ajouter le **caractère** 'c' au contenu de la variable var qui contient déjà la **valeur entière** 16.
Cela est incompatible et l'interpréteur, essayant de réaliser l'addition, nous signale une erreur.

Nous venons déjà de parler de deux types existants dans le langage Python : le type **entier**, le type **chaines de caractères**.

Le type entier est un type simple : une valeur de type entier ne représente qu'une seule valeur.

Le type chaine de caractères (ou *string* en anglais) est un type complexe: une valeur de ce type contient plusieurs caractères qui peuvent être adressés individuellement.

Un autre type simple est le type flottant qui représente un nombre réel. Les nombres réels peuvent être représentés de différentes manières :

.1416
.023742e2 (notation avec mantisse et exposant : $0{,}023742 \times 10^{2}$)
31416e-4 (31416×10^{-4})

II.2.3 Les chaines de caractères

Les chaines de caractères sont des valeurs composées de plusieurs caractères. Il n'y a pas en Python de type 'caractère individuel' comme en langage C par exemple (mais on peut très bien avoir une chaine ne comportant qu'un seul caractère).

Voici des exemples de chaines littérales :

'Bonjour'
"Bonjour"
'Une chaine avec un retour à la ligne\n'

Les chaines sont un exemple de **collection** dont nous verrons d'autres exemples par la suite. Les collections permettent le stockage et la manipulation en mémoire d'ensembles de données représentés par des variables.

Pour adresser (accéder à) un élément particulier d'une collection, on dispose de l'opérateur **[]**
par exemple :

```
myString = "ma chaine"
print myString
print myString[4]
```

Remarque : le premier indice est 0. Ainsi, le premier caractère est myString[0] et le dernier est myString[*la taille-1*]

Pour manipuler une collection telle qu'une chaine de caractères, nous disposons aussi de **fonctions** et d'**opérateurs** attachés au nom des variables.

Par exemple pour connaître la taille d'une chaine de caractères :

```
taille = len(myString)
print taille
myUpperString = myString.upper()
```

Nous découvrons ici des **fonctions** (len, upper) appartenant à la librairie standard de Python.

Nous reviendrons plus en détail dans la partie III de ce cours sur les collections, dont les chaines de caractères sont un cas particulier.

[String Methods : http://docs.python.org/2.7/library/stdtypes.html#string-methods]

II.2.4 Le formatage des chaines de caractères

Nous pouvons, pour l'affichage d'informations à l'écran par exemple, avoir besoin de créer des messages comportant des informations de différents types, comme des caractères et des nombres.
Or nous avons vu précédemment qu'il n'est pas possible de « concaténer » directement des valeurs de types différents.

Nous devrons alors utiliser des fonctions de conversion de type.

Exemple :

```
prix = 16
message = "Le prix est " + str(prix) + " euros"
print message
```

Pour faciliter ce genre d'opérations très courantes, le langage Python dispose d'un opérateur spécial de formatage des chaines de caractères : %
La syntaxe générale est :

```
chaine_avec_formats % valeurs
```

la spécification des formats eux-mêmes dans la chaine fait appel à l'opérateur %. Ainsi %d signifie que l'on va placer dans la chaine une valeur de type décimal. Notre exemple devient ainsi :

```
prix  = 16
message = "Le prix est %d euros" % prix
print message
```

Autre exemple :

```
val1 = 5
val2 = 9
message = "La somme de %d et %d vaut %d" % (val1, val2, val1+val2)
```

Les dernières versions de Python ont introduit une méthode que l'on peut préférer à l'opérateur % pour le formatage des chaines de caractères. Il s'agit de la fonction *format* qui fait désormais partie des objets de type chaine de caractères. Voici sur un exemple :

```
article = "stylos"
quantite = 5
prix = 4,5
message = "le prix de {0} {1} est {2}".format(quantite, article, prix)
print message
```

Cela affiche :

```
le prix de 5 stylos est 4.5
```

[formatage des chaines : http://docs.python.org/2.7/library/stdtypes.html#string-formatting-operations]

II.2.5 Les Entrées / Sorties

Un programme doit avoir des moyens de communiquer avec l'extérieur de la seule mémoire vive de l'ordinateur. Les opérations qui visent à lire des données et à en écrire ailleurs qu'en mémoire vive sont des **opérations d'entrées / sorties**.

Parmi les opérations d'entrées / sorties les plus courantes figurent la lecture de caractères au clavier (entrée) et l'écriture d'information à l'écran (sortie).

Il y en a d'autres, également très courantes, comme lire et écrire dans des fichiers sur le disque dur, recevoir et envoyer des données par l'intermédiaire d'une interface réseau.

Pour lire des caractères au clavier le langage Python fournit la fonction : **raw_input**

```
chaine = raw_input("Saisissez des caractères puis tapez 'Entrée' : ")
```

La fonction raw_input prend une chaine en argument (un message qui s'affiche à l'écran) et retourne la chaine saisie au clavier (la saisie se terminant par la frappe de la touche "Entrée").
Nous reverrons en détail les fonctions, leurs arguments et leur valeur retournée.

Ici, ce qui a été saisi au clavier est placé, sous la forme d'une chaine de caractères, dans la variable nommée chaine .

Pour les affichages à l'écran, le langage Python offre une instruction spéciale : **print**

```
print chaine
```

La chaine à afficher peut contenir des ordres spéciaux en plus des caractères à afficher à l'écran.Ces «ordres spéciaux» se présentent sous la forme de «séquences d'échapement» ('escape sequence' en anglais) est son introduite par le caractère \ (back slash).

Par exemple :

\n pour passer à la ligne
\' permet d'afficher la simple cote qui sinon est interprétée comme délimiteur de chaine
\" permet d'afficher la double cote, sinon interprétée elle aussi comme délimiteur de chaine

\hxx permet de spécifier un caractère ASCII (codé sur 8 bits) en fournissant sa valeur hexadécimale (chaque x représente un digit hexadécimal : valeur de 0 à F - soit 0 à 15 en décimal - correspondant à 4 bits. xx représente donc une valeur codée sur 8 bits)

\uxxxx représente le codage hexadécimal d'un caractère unicode codé sur 2 octets (1- bits). Cette séquence d'échappement ne peut être utilisée qu'avec les chaines de caractères unicode. Une chaine unicode s'écrit avec le préfixe u, par ex :

print u"cha\u00eene unicode"

II.2.6 Les opérateurs

Les opérateurs permettent de manipuler les variables en mémoire pour obtenir des résultats.

Le premier opérateur est **l'affectation représenté par le signe** =. Cet opérateur fondamental, qu'il ne faut pas confondre avec le test d'égalité mathématique que nous verrons ultérieurement, a déjà été présenté.

a/ Opérateurs arithmétiques :

+	addition	a = b+5 # affecte dans a la valeur représentée par b+5
-	soustraction	c = c – f # soustrait de c la valeur de f
*	multiplication	
/	division	a = 5/2 # la division ici est une division entière car les deux opérandes le sont. Le résulatat est donc la valeur entière 2 a = 5.0 / 2 # division décimale car un des opérateur est décimal. Le résultat sera 2.5
%	opérateur modulo : reste de la division entière	a = 5 % 2 # a est affecté avec le reste de la division de 5 par 2, ce qui donne la valeur 1
+=	ajout d'une quantité à une variable	a += 4 # on ajoute 4 à la valeur (supposée numérique) contenue dans la variable a
-=	retrait d'une quantité à une variable	a -= 4 # on ajoute 4 à la valeur (supposée numérique) contenue dans la variable a

b/ Opérateurs logiques

Les opérateurs logiques permettent de combiner des **valeurs booléennes**, c'est à dire des éléments dont la valeur est interprétée comme **vrai** ou **faux**, pour obtenir des expressions logiques plus complexes.

Pour obtenir la liste exhaustive des opérateurs logiques, on se reportera à la documentation officielle de Python. Ce sera un bon exercice de recherche et de familiarisation avec cette documentation. C'est une source exhaustive à laquelle un programmeur Python a régulièrement besoin de se référer. On accède avec cette documentation grâce au menu Help de la fenêtre *Python Shell* du *Python IDLE* (raccourci clavier par F1). Cela nous envois sur le site :
http://docs.python.org/2.7/

Les opérateurs logiques sont décrits dans la partie *The Python Standard Librar*y, au paragraphe 5.3

Dans les expressions arithmétiques et logiques complexes, il faut savoir quelles opérations sont réalisées en premier. Cela est indispensable pour déterminer le résultat de toute expression composée. Voici dans la documentation Python le tableau de précédence (priorité) des opérateurs :
http://docs.python.org/2/reference/expressions.html#operator-precedence

c/ Opérateurs sur les chaines de caractères

Se reporter au tableau d'**opérations** du paragraphe 5.6 de *The Python Standard Library* dans la documentation Python présentée précédemment.
On regardera aussi les **méthodes** applicables aux **objets string** dans le paragraphe 5.6.1 de la documentation Python.

[String operations and methods :
http://docs.python.org/2.7/library/stdtypes.html#sequence-types-str-unicode-list-tuple-bytearray-buffer-xrange]

II.2.7 Les valeurs booléennes et les tests

On appel « valeur booléenne » une valeur qui représente une évaluation logique et qui représente donc l'une des deux possibilités suivantes : vrai ou faux.

Les valeurs booléennes sont le résultat de l'évaluation d'**expressions logiques** et elles servent à faire des choix dans un programme (effectuer telle action quand telle condition est réalisée).

Voici quelques exemples d'expressions logiques à partir de deux variables a et b dont nous supposons qu'elles contiennent des valeurs entières.

Expressions logiques simples :

a > 5
a <= 10
a < b
a == b (égalité logique entre a et b)

Expressions logiques composées :
a >= 5 and a <= 10 (cette expression est vraie si a est compris entre 5 et 10 inclus)
not (a < 5 or a > 10) (cette expression a la même valeur de vérité que la précédente)

Toutes ces expressions représentent une **valeur booléenne**, elles sont soient vraies, soit fausses.

[Comparisons : *The Python Standard Library*, au paragraphe 5.3]

II.2.7.1 La structure if

Les expressions logiques, ou les valeurs booléennes, peuvent être utilisées dans **des instructions conditionnelles** introduite par le mot réservé **if**.

Voici la structure d'un bloc if :

```
if  expression_logique :
        instruction 1
        instruction 2
        .
        .
        Instruction n
else :
        autre_bloc_instructions
```

Si l'expression logique est évaluée à vrai le **bloc d'instruction** sous le test est exécuté, sinon, si l'expression logique est évaluée à faux, c'est le bloc d'instruction sous le mot réservé else qui est exécuté.

On constate qu'en langage Python, les blocs d'instructions sont **définis grâce au décalage** (ou indentation) des instructions qui le composent. Cela est propre à Python, d'autres langages, comme le C, utilisent des marqueurs plus explicites ({ bloc instructions }). L'avantage de rendre le décalage obligatoire comme marqueur de bloc d'instructions (par ailleurs conseillé dans tous les langages) est qu'il oblige à une écriture claire des programmes.

Remarque : Dans la structure if, l'alternative else est facultative. Dans ce cas, si l'expression logique est évaluée à faux, le bloc d'instruction sous le if est simplement sauté, l'exécution du programme reprenant à la première instruction après le bloc if (donc en fait la première instruction qui aura le même décalage que le if lui-même).

Il y a un autre mot réservé **elif**, qui permet des instructions de tests successifs, ce qu'on appelle aussi l'alternative multiple

Exemple :

```
temperature = getTemperature()
if temperature < 0 :
        print "il gèle"
elif temperature >=0 and temperature <100 :
        print "l'eau est liquide"
else :
        print "l'eau est à l'état gazeux"
```

Le résultat d'une expression logique peut être placé dans une variable. Une telle variable prend alors une **valeur booléenne**.

Par exemple :

```
estLiquide = (temperature >=0 and temperature <100)
if estLiquide :
        print "l'eau est à l'état liquide"
```

Le langage Python fournit les deux valeurs **True** et **False** pour représenter les valeurs booléennes. Mais comme en langage C, toute valeur peut être interprétée comme une valeur booléenne de la manière suivante : **si une variable contient la valeur 0 ou une chaine (de caractères) vide, elle est interprétée comme fausse, toute variable qui ne contient pas 0 ou toute chaine non vide est interprétée comme vrai**.

II.2.8 Les structures de contrôle ou boucles

II.2.8.1 La boucle while

Nous venons de voir comment tester une expression logique ou valeur booléenne avec la structure if.

Il est aussi possible d’exécuter un même bloc d’instructions tant qu’une condition est vérifiée grâce à la structure **while** dont voici la syntaxe :

```
while condition :
        bloc_instructions
```

Comme dans la structure if, la condition est d’abord évaluée et si elle vraie, le bloc d’instructions est exécuté. Mais ici, après exécution du bloc d’instruction, la condition est évaluée à nouveau et le bloc d’instructions est ré-exécuté tant que la condition est vrai. Quand la condition devient fausse, le contrôle du programme passe à l’instruction suivant le bloc while.
Le bloc d’instruction est, ici également, défini par son **indentation ou décalage**

Remarque : il est très important, dans la structure while, que le bloc d’instructions ait une **influence sur la condition,** de manière à ce que si la condition est vraie au départ, elle puisse devenir fausse après un certain nombre de tours dans la boucle, sans quoi on se trouverait dans le cas d’une **boucle infinie e**t d’un programme bloqué.

Exemple :

```
temperature = 25
while temperature <= 100 :
        temperature += 1
        print "la température est %d, l'eau est liquide\n" % temperature

print "le chauffage de l’eau est terminé, la temperature est %d\n" % temperature
```

II.2.8.2 La boucle for

Il y a aussi en langage Python une structure **for** permettant de parcourir un nombre fini d'éléments.
La structure générale de la boucle for est la suivante :

```
for element in tableau :
        bloc instructions
```

Dans le bloc d'instructions, le programme dispose de la variable représentant **l'élément courant pris dans le tableau.** Ci-dessus cette variable a été nommée element, mais tout autre nom correct de variable convient. Le bloc d'instructions sera exécuté autant de fois qu'il y a de valeur dans le tableau, chaque valeur étant placée tour à tour dans la variable élément.
Les mots **for** et **in** sont les deux mots réservés utilisés dans cette structure.

Exemple :
Nous n'avons pas encore étudié les tableaux ou collections d'éléments en Python (il y en a plusieurs types et ces types sont une des forces du langage Python) mais d'ores et déjà nous pouvons utiliser la fonction **range(val1, val2)** de la bibliothèque standard de Python. Cette fonction renvoie la « collection » des valeurs comprises entre val1 (inclus) et val2 (exclus).

Par exemple range(1,10) renvoie la liste des 9 éléments suivants :
[1,2,3,4,5,6,7,8,9]
Nous verrons ultérieurement que cette écriture correspond à une **liste modifiable**.

Voici comment utiliser une boucle for pour afficher toutes les valeurs d'un intervalle :

```
for i in range(5,11) :
        print i
```

Cela est équivalent à la boucle suivante

```
for i in [5,6,7,8,9,10] :
        print i
```

Ou encore, il est possible d'utiliser une liste préalablement placée dans une variable :

```
maListe = [5,6,7,8,9,10]
for i in maListe :
        print i
```

II.2.8.3 Les instructions *break* et *continue*

Les mots réservés **break** et **continue** sont utilisés pour modifier le cours normal du déroulement d'une boucle for ou d'une boucle while.

Il arrive en effet que dans certaines situations (détectées par un test if à l'intérieur d'une boucle), on décide soir de sortir de la boucle (grâce à l'instruction **break**), soit de sauter le reste des instructions de la boucle pour commencer directement l'itération suivante de la boucle (grâce à l'instruction **continue**).

[break et continue :
http://docs.python.org/2.7/tutorial/controlflow.html#break-and-continue-statements-and-else-clauses-on-loops]

II.2.9 Un premier cas de collection d'éléments : les listes

Nous venons d'introduire les listes dans l'exemple précédent.
Tous les langages de programmation offrent, d'une manière ou d'une autre, la possibilité de manipuler des **collections d'éléments**, souvent appelées tableaux.

Un tableau est généralement une collection d'éléments de même type. Par exemple en langage C, on déclare un tableau d'entiers de la manière suivante :

```
int monTableau[10] ; // déclare un tableau de 10 entiers
                     // et réserve la place nécessaire en mémoire pour contenir les 10 entiers
```

l'accès aux éléments du tableau se fait ensuite en utilisant le nom du tableau et en fournissant un **indice entre crochets** :

```
monTableau[5] = 150 ; // placer la valeur 150 dans le tableau à la position d'indice 5
val = monTableau[0] ; // place dans une variable appelée val, la valeur contenue à l'indice 0
                      // du tableau.
```

En langage C, le premier indice du tableau est toujours 0 et le dernier indice est donc sa taille – 1.

> Le programmeur doit toujours avoir une idée très précise des limites d'un tableau et de la correspondance des indices sans quoi il s'expose à des erreurs de programmation (effets de bords).

En langage Python, les **listes** fonctionnent un peu comme les tableaux du langage C mais avec quelques différences notables (qui facilitent grandement la vie du programmeur!) :

comme pour toute autre variable en Python, on créé une liste sans déclaration préalable, sans préciser le type des éléments contenus dans la liste.

maListe = [1,2,3,4,5,6,7,8,9] # créé 'ex-nihilo' une liste de 9 valeurs entières

on peut aussi créé une liste vide au départ :

maListe = []

ou encore :

maListe = list()

La taille d'une liste n'est pas figée après sa première création. On pourra toujours ajouter des éléments. C'est l'interpréteur Python qui se charge tout seul de la gestion de la mémoire nécessaire pour contenir et gérer la liste.

maListe.append[99] # ajoute la valeur 99 comme nouvel élément à la fin de la liste

print maListe # affiche [1,2,3,4,5,6,7,8,9,99]

Contrairement aux langages traditionnels comme le C, il est tout à fait possible de placer dans une même liste des éléments de type (très) différents. Par exemple une même liste peut contenir non seulement des nombres entiers et des nombres flottants mais on peut aussi y ajouter des chaines de caractères.

monAutreListe = [52, "bonjour", "1", 2]

Un éléments commun à C et Python est la **notation à crochets** pour adresser des éléments individuels des listes.

Par exemple :

```
uneListe = [5,7,9]
val = uneListe[1] + uneListe[2]

#val contiendra la valeur 7+9 càd 16
```

Et tout comme en langage C, le premier indice d'une liste en Python est 0. Le dernier indice est donc la taille de la liste (nombre d'éléments) moins 1.

Pour connaître le nombre d'éléments contenus dans une liste, la bibliothèque standard de Python fournit la fonction len.

Par exemple :
taille = len(uneListe)

Nous verrons ultérieurement que les listes fournissent plusieurs méthodes permettant de les manipuler ou d'en extraire des sous listes.
D'ailleurs les listes ne sont qu'un cas particulier des différents types de collections fournit par le langage Python. Nous en avons déjà rencontré un autre cas : les chaines de caractères. Il y en a encore d'autres que nous étudierons par la suite.

Rappel : nous avons vu au paragraphe précédent II.2.8 qu'il existe une structure de contrôle, la boucle **for.... in** qui permet de parcourir les éléments d'un tableau, l'un après l'autre.

[Lists : http://docs.python.org/2.7/tutorial/introduction.html#lists]

II.2.10 Les sous-programmes ou fonctions

II.2.10.1 Présentation

Dès qu'un programme atteint une certaine taille, il est nécessaire de le découper et de le séparer en plusieurs parties correspondant chacune à un traitement plus spécifique. Cette découpe fonctionnelle facilite la lecture et la maintenance des programmes et permet de réutiliser certaines portions de traitement sans qu'il soit besoin de les réécrire plusieurs fois.

Par exemple les méthodes vues précédemment pour manipuler les chaines de caractères sont des sous-programmes offerts au programmeur Python par la *Python Standard Library*.

Nous avons déjà rencontrés des fonctions dans les exemples des chapitres précédents, par exemple des fonctions pour manipuler des chaines de caractères.

Reprenons l'exemple très simple vu en II.2.3 :

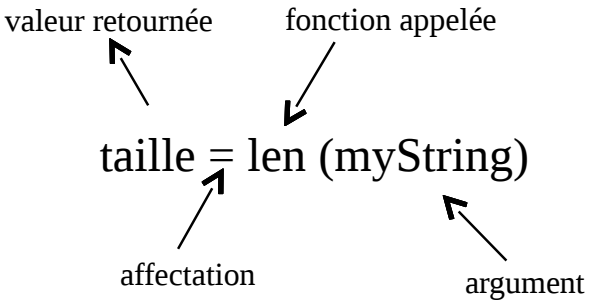
```
myString = "ma chaine"
print myString
print myString[4]
taille = len(myString)
print taille
myUpperString = myString.upper()
```

Nous avons ici deux fonctions (appartenant la la bibliothèque standard de Python) :

len() et upper()

la première est une fonction autonome qui admet **un argument donné entre parenthèses**. C'est la fonction len(). On écrit toujours une fonction en indiquant un jeu de parenthèse qui symbolise le passage d'argument, même pour les fonctions qui n'en prennent aucun (liste d'arguments vide).

La fonction len() prend une chaine de caractère comme argument et elle effectue le calcul du nombre de caractères contenus dans cette chaine. Elle renvoie ensuite la valeur calculée dans une variable du programme :



L'autre fonction rencontrée dans cet exemple est la fonction upper(). Celle-ci fonctionne de manière un peu différente.
Elle s'applique aussi à une chaine de caractères et elle a pour but de créer une nouvelle chaine de caractères : une copie de la première chaine mais avec tous les caractères passés en majuscules.
La chaine de départ n'est pas ici donnée en argument. La fonction, ici, fait partie de l'objet "chaine de caractères" et l'appel est réalisé de la manière suivante :

myUpperString = myString.upper()

Donc la fonction s'applique directement à la variable chaine par l'opérateur **point** (**.**)
En fait, et nous verrons cela en détail par la suite, une chaine de caractère en Python est un **objet** qui contient à la fois des données (les caractères de la chaine) et des traitements (les fonctions qui s'appliquent à la chaine elle-même)

Ici la chaine s' applique la fonction upper(), ce qui a pour effet de renvoyer dans une variable du programme (ici la variable myUpperString) une copie de la chaine où tous les caractères ont été convertis en majuscule.

II.2.10.2 Définition d'une fonction

On utilise le mot réservé **def** pour créer un sous-programme (appelé aussi *fonction*) et on indique entre parenthèses les **arguments** avec lesquels ce sous-programme devra travailler. Les arguments sont les valeurs sur lesquelles on veut effectuer certains traitements.

La forme générale de définition d'une fonction est la suivante :

def NomDeLaFonction (arg1, arg2, …) **:**

```
    instruction1
    instruction 2
    ...
    return valeur
```

Le mot réservé **return** permet à la fonction le renvoie d'une valeur au **programme appelant**. L'utilisation est facultative car il est possible d'écrire des sous-programme réalisant des traitement mais ne retournant pas de valeur.

Exemple :
Dans l'exemple précédent nous avons vu l'existence d'une fonction **len** fournie par la bibliothèque standard de Python pour calculer la longueur d'une chaine de caractères. Voici comment on pourrait écrire cette fonction, mais nous allons lui donner un autre nom pour ne pas entrer en conflit avec le symbole len déjà existant :

```
def tailleChaine(laChaine) :
    taille = 0
    for c in laChaine :
        taille += 1
    return taille
```

II.2.10.3 Utilisation d'une fonction

Pour invoquer une fonction depuis un **programme appelant** (la fonction est alors le **programme appelé**) on écrit son nom suivi par la liste des valeurs qui vont correspondre aux arguments déclarés lors de la définition de la fonction. Les valeurs sont écrites entre parenthèses.

Exemple :

Nous allons appeler la fonction de l'exemple précédent pour l'appliquer à une chaine de caractère concrète.

```
maChaine = "Bonjour"
tail = tailleChaine(maChaine)
print "la taille de la chaine \"%s\" est %d \n" % (maChaine, tail)
```

II.2.10.4 Retour sur le passage d'argument à une fonction

Il y a deux manières de passer des arguments à une fonction : par **valeur** ou par **référence**.
La question qui se pose est la suivante : les modifications faites sur les arguments à l'intérieur de la fonction sont ils répercutés ou non à l' extérieur de la fonction ?
En langage Python, les arguments sont passés par **valeur**. Python créé une table de symboles locale à chaque entrée dans une fonction et cette table contiendra une copie provisoire des variables locales et des arguments formels. La valeur passée ne consiste cependant pas en une recopie des objets dans la table de symboles locale, mais en une copie de la référence de l'objet.
Ainsi, pour les types de données modifiables (*mutable objects* dans la documentation Python), comme les listes, c'est bien une référence à la liste qui est passée et si l'on altère un des éléments de la liste, cette altération sera effective même après le retour dans le programme appelant.

[Les fonctions dans la doc Python : http://docs.python.org/2.7/tutorial/controlflow.html#defining-functions]

II.2.11 La gestion des erreurs d'exécution, les exceptions

Rappel : Quand l'interpréteur Python rencontre une instruction qu'il n'est pas en mesure d'exécuter (erreur de syntaxe, division par zéro, accès à un fichier qui n'existe pas, etc), il **lance une exception.** Si cette exception n'est pas gérée par le programme dont l'instruction fait partie, le résultat est l'arrêt du programme avec affichage d'un message d'erreur.

Par exemple, voici une instruction demandant la saisie d'un nombre flottant. L'instruction utilise la fonction raw_input() qui lit une chaine de caractères au clavier, puis la fonction float() tente de convertir cette chaine en un nombre flottant :

```
val =float(raw_input("saisir un flottant : "))
```

si l'utilisateur saisit des caractères qui ne peuvent pas être convertis en un nombre flottant, nous aurons une exception puis un message d'erreur :

```
>>> val =float(raw_input("saisir un flottant : "))
saisir un flottant : 5.5.2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    val =float(raw_input("saisir un flottant : "))
ValueError: invalid literal for float(): 5.5.2
```

Or il y a moyen pour le programmeur de contrôler les instructions pouvant générer des exceptions et de définir ses propres traitements d'erreur. Pour cela le langage Python offre le mécanisme très efficace des blocs **try** et **except**.
Les instructions pouvant générer des exceptions sont placées dans un bloc **try** et un ou plusieurs blocs **except** servent à définir les traitements d'erreur.
Les exceptions générées par les fonctions de la bibliothèque Python ont des identifiants qui permettent de les distinguer et d'effectuer les traitements adéquats.

Exemples:

1/ Reprenons l'exemple précédent de la saisie de caractères au clavier pour les transformer en un nombre flottant. On peut utiliser un bloc **try... except...** dans une boucle pour redemander la saisie jusqu'à obtenir une chaine convertible en nombre flottant :

```
saisie_ok = False
while (not saisie_ok) :
   try :
       reel1 = float(raw_input("nombre 1 : "))
       saisie_ok = True
   except ValueError:
       print ("ce n'est pas un flottant. Recommencez")
       saisie_ok = False
```

2/ Voici un autre exemple concernant l'ouverture d'un fichier. Si le fichier n'est pas trouvé à l'endroit indiqué par la fonction open, la fonction open génère une exception IOError. La gestion de cette exception permet l'affichage d'un message adéquat :

```
try :
       nom = raw_input("Entrez un nom de fichier : ")
       fic = open(nom)
       data = fic.read()
       fic.close()
       break
except IOError:
       print("erreur lecture du fichier")
       sys.exit()
```

Remarque : Les exceptions ValueError et IOError rencontrées dans les exemples précédents sont générées respectivement par les fonctions raw_imput() et float() de la bibliothèque standard de

Python. Il est également possible pour le programmeur de déclencher explicitement une exception grâce à l'instruction **raise** <Nom d'une Exception>, et aussi il est possible de définir de nouvelles exceptions pour des besoins particuliers, mais cela sort du cadre de ce cours.

Pour en savoir plus :

[Errors and Exceptions : http://docs.python.org/2.7/tutorial/errors.html]

Et pour connaître la liste des exceptions définies dans la bibliothèque standard de Python :

[Built-in Exceptions : http://docs.python.org/2.7/tutorial/errors.html]