3 - Types de données

1. Les données numériques et booléennes
2. Les chaînes de caractères
3. Les listes
4. Les tuples
5. Les dictionnaires

3.1 - Les données numériques et booléennes

- Les principaux types numériques sont **: int** (entier), **long** (entier long), **float** (flottant).

- Si deux expressions sont de types numériques différents et si elles sont combinées à l'aide d'un opérateur, celle qui est dans le type le plus simple est convertie dans le type le plus complexe.

- Les expressions booléennes ont le type **bool** et deux valeurs possibles : **True** et **False**

- Une expression booléenne peut être combinée avec une expression numérique à l'aide d'un opérateur numérique. Sa valeur est alors convertie en 0 ou 1.

3.1 - Les données numériques et booléennes

```
>>> a = 1
>>> while a < 5 :
    print a**a**a , "    " , type(a**a**a)
    a += 1
>>>

1       <type 'int'>

16       <type 'int'>

7625597484987       <type 'long'>

13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084096       <type 'long'>

>>>
>>> a = 1.0
>>> while a < 5 :
    print a**a**a , "    " , type(a**a**a)
    a += 1
>>>

1.0       <type 'float'>

16.0       <type 'float'>

7.62559748499e+12       <type 'float'>

1.34078079299e+154       <type 'float'>
>>>

>>> (2+False)*(2+True)

6

>>>
```

3.2 - Les chaînes de caractères

- Une chaîne de caractères est une suite de caractères délimitée par des **apostrophes** (simple quotes) ou des **guillemets** (double quotes). C'est une constante de type **Str**.

- Quand au sein d'une chaîne de caractères, on veut insérer des caractères spéciaux, on les fait précéder du caractère **antislash** \ (backslash): \n représente un saut à la ligne, \t une tabulation, \' une apostrophe, \" des guillemets. Le caractère \ suivi d'un saut à la ligne permet d'écrire une chaîne de caractères sur plusieurs lignes.

- Pour entrer une chaîne de caractères, dans laquelle il y a des caractères de saut à la ligne, on peut la mettre entre **triples guillemets** ou **triples apostrophes**.

3.2 - Les chaînes de caractères

- On peut convertir une expression d'un type quelconque en une chaîne de caractères en la mettant entre **backquotes** (`). L'interprète calcule la valeur de l'expression et c'est valeur qui est ensuite convertie en chaîne de caractères.

- Pour représenter une chaîne de caractères sous forme d'une suite de leurs codes UTF8 (norme **Unicode**), on fait précéder cette chaîne du symbole *u* (par exemple *u'bonjour'*). La chaîne prend alors le type *unicode* qui est différent de *str*.

3.2 - Les chaînes de caractères

```
>>> x='l\'eau\n\"Perrier\" '
>>> x
'l\'eau\n"Perrier" '

>>> print 'l\'eau\n\"Perrier\" '
l'eau
"Perrier"

>>> print 'l'eau \
\"Perrier\"'
l'eau"Perrier"

>>> print '''l'eau
    \"Perrier\"'''
l'eau
"Perrier"
```

```
>>> y =2
>>> z = `y+1`
>>> print y+1, type(y+1), z, type(z)
3 <type 'int'> 3 <type 'str'>

>>> for c in 'déjà' :
    print c
d
?
?
j
?
?

>>> for c in u'déjà' :
    print c
d
é
j
à
>>>
```

3.2 - Les chaînes de caractères

- Les chaînes de caractères comme les nombres font partie d'un type de constantes, les **littéraux**, mais contrairement à ces derniers, les chaînes de caractères sont des données complexes rassemblant en une seule entité des données plus simples, les caractères.

- On peut accéder au caractère situé à la position *n* dans la chaîne de caractères *s* à l'aide de l'expression *s[n].* La position du premier caractère d'une chaîne est la position *0*. On peut aussi y accéder à partir de la fin à l'aide de l'expression *s[-m], m* étant le nombre de caractères comptés en allant du caractère concerné jusqu'au dernier, les deux étant inclus dans le comptage.

3.2 - Les chaînes de caractères

- On peut accéder à une sous-chaîne d'une chaîne *s* (tranchage ou **slicing**) à l'aide de l'expression *s[n:m]*. La valeur de cette expression est la sous-chaîne de *s* allant de la position *n* à la position *m-1*. Si $m \leq n$, alors la sous-chaîne est la chaîne vide.

- Pour concaténer deux chaînes de caractères, on peut utiliser l'opérateur binaire +. Pour répéter une chaîne de caractères, on peut la multiplier par une entier à l'aide de l'opérateur *.

3.2 - Les chaînes de caractères

```
>>> print ‘bonjour’[2]
n
>>> print ‘bonjour’[-2]
u
>>> ‘bonjour’[2:5]
‘njo’
>>> ‘bonjour’[2:-2]
‘njo’
>>> ‘bonjour’[2:2]
‘’
```

```
>>>s='Je vais. Je viens'
>>>fin = False
>>>k = 0
>>>while not fin :
    if s[k:k+2] == '. ' :
        if s[k+2] >= 'A' and s[k+2] <= 'Z' :
            fin = True
    k += 1
print s[0:k]

Je vais.

>>>
```

3.2 - Les chaînes de caractères

- Pour traiter les chaînes de caractères, Python fournit des **fonctions prédéfinies**. Une fonction est un programme qu'on peut appeler à l'aide du nom de la fonction suivi entre parenthèses de ses arguments. Elle retourne alors une valeur.

- Le tableau ci-dessous présente les principales fonctions de manipulation de chaînes de caractères. La liste des fonctions prédéfinies est sur le Web à l'adresse suivante :

  http://docs.python.org/lib/built-in-funcs.html

Appel de fonction	Signification
len(s)	retourne la longueur de la chaîne de caractères s
chr(n)	retourne le caractère dont le code ASCII est l'entier n
ord(c)	retourne l'entier qui est le code ASCII du caractère c
unichr(n)	retourne le caractère dont le code UTF8 est l'entier n

3.2 - Les chaînes de caractères

```
>>>entree = u'Je vais. Je viens'
>>>sortie = u”
>>>for c in entree :
    if c >= u’a' and c <= u’z' :
        code = ord(c )
        sortie += unichr(code-
    32)
    else :
        sortie += c
print sortie

JE VAIS. JE VIENS

>>>
```

```
>>>entree = u'Je vais. Je viens'
>>>sortie = u”
>>> longueur = len(entree)
>>>for k in range(longueur):
    c = entree[k]
    if c >= u’a'  and <= u’z' :
        code = ord(c )
        sortie += unichr(code-
    32)
    else :
        sortie += c
print sortie

JE VAIS. JE VIENS

>>>
```

3.2 - Les chaînes de caractères

- Les chaînes de caractères , comme **objets** Python, sont associés à des fonctions liés à leur type, qu'on appelle des **méthodes**. Pour un objet donné *x*, il est possible de connaître les méthodes associées grâce à l'appel de fonction *dir(x).*

- L'appel d'une méthode *m* associée à l'objet x avec les arguments $a_1\ a_2\ \ldots\ a_n$, s'effectue par l'expression $x.m(a_1, a_2, \ldots a_n)$.

- Le tableau ci-dessous présente quelques méthodes associées aux chaînes de caractères (voir la liste exhaustive sur le Web à l'adresse : *http://docs.python.org/lib/string-methods.html*)

Méthode	Signification
split(sep)	retourne la liste de toutes les sous-chaînes obtenues par découpage de la chaîne à l'aide du séparateur *sep* (s'il est omis, le séparateur par défaut est un séparateur standard)
find(subs, debut, fin)	retourne la position de la première occurrence de la sous-chaîne *subs* trouvée dans le segment *[debut : fin]* de la chaîne. Retourne -1 si aucune occurrence n'est trouvée. Les deux derniers arguments sont optionnels.

3.2 - Les chaînes de caractères

```
>>>phrase = u'J\'ai mangé de l\'époisse, de la tome et du comté'
>>>mot = u'tome'
>>>trouve = False
>>>for m in phrase.split() :
    if m == mot :
        trouve = True
        break
if trouve :
    print 'le mot \" ', mot, '\" est dans la phrase '\" , phrase, '\"  '
else :
    print 'le mot \" ', mot, '\" n\'est pas dans la phrase \ \"', phrase, '\" '

Le mot " tome " est dans la phrase " J'ai mangé de l'époisse, de la tome et
    du comté "
>>>
```

3.2 - Les chaînes de caractères

1. Donner le résultat de l'exécution du programme ci-dessous lorsque l'on entre au clavier la valeur "tout" pour *mot*. Même question lorsque l'on entre "tot". En déduire le rôle de ce programme.

```
mot = input("Entrez un mot !")
longueur = len(mot)

correct = True

k = 0

while correct and k < (longueur+1)/2:
    if mot[k] != mot[-k-1] :

        correct = False

    k += 1

print  correct
```

3.2 - Les chaînes de caractères

2. Donner le résultat de l'exécution du programme ci-dessous lorsque l'on entre au clavier le texte "Il mange souvent lentement mais digère rapidement. Il faut savoir qu'il ne ment jamais." .En déduire le rôle de ce programme.

```
texte = input("Entrez un texte ! ")
liste_mots = texte.split()
k = 0
for mot in liste_mots:
    position = mot.find('ment')
            if position + 4 == len(mot) :
                k += 1
print  k
```

3.2 - Les chaînes de caractères

3. Ecrire un programme Python pour chacune des spécifications ci-dessous.

a) Compter le nombre d'occurrences d'un caractère donné dans un texte.

b) Compter le nombre d'occurrences d'un mot donné dans un texte (on suppose que les séparateurs de mots sont ceux utilisés par la méthode split)

c) Remplacer un mot par un autre dans un texte (même définition des séparateurs de mots qu'à l'exercice précédent).

d) Extraire d'un texte la liste de tous les mots commençant par une majuscule.

e) Extraire d'un texte la liste de toutes les valeurs numériques (pour les nombres décimaux, la virgule est représentée par un point et on admet la notation scientifique où la mantisse, qui n'est pas vide, est séparée de l'exposant par un « e » ou « E »)

Exemple: si le texte est "122.44 abbEcaa6.4E7666E90.5", en sortie le programme renvoie la liste [122.44, 6.4E7666, 90.5].

3.3 - Les listes

- Les listes sont des suites d'objets Python qui peuvent être hétéroclites. Elles ont le type **list.** Leur syntaxe est la suivante : $[\, o_1, o_2, \ldots, o_n \,]$

- Comme les chaînes de caractères , les listes sont des objets complexes du type **sequence**. On peut donc leur appliquer les opérateurs + et *. De même, on peut accèder à un élément d'une liste, ainsi qu'à une sous-liste par l'opération de tranchage.

- Mais à la différence des chaînes de caractères, ce sont des objets modifiables (**mutable**) : on peut modifier un élément de la liste alors que l'on en peut pas modifier un caractère d'une chaîne. Cela permet d'utiliser les opérations d'accès à un élément et de tranchage pour modifier une liste.

3.3 - Les listes

```
>>>liste = [23, ‘bonjour’, [1,2]]
>>>print liste[2]
[1,2]
>>>print liste[2][0]
1
>>>liste[1] = ‘bonsoir’
>>>print liste
[23, ‘bonsoir’, [1,2]]
>>> liste[1:1] = [‘je’, ‘dis’]
>>>print liste
[23, ‘je’, ‘dis’, ‘bonsoir’, [1,2]]
>>> liste += [3,4]
>>>print liste
[23, ‘je’, ‘dis’, ‘bonsoir’, [1,2], 3, 4]
```

```
>>>texte = u’Je suis tombé malade. Je n’ai pas travaillé.’
>>> liste_mots = texte.split()
>>> liste_mots_etiquetes = []
>>> for mot in liste_mots :
      liste_mots_etiquetes += [[mot, len(mot)]]

>>> print liste_mots_etiquetes
[[u'Je', 2], [u'suis', 4], [u'tomb\xe9', 5], [u'malade.', 7], [u'Je', 2],
      [u'n\u2019ai', 4], [u'pas', 3], [u'travaill\xe9.', 10]]
```

3.3 - Les listes

- Les listes, comme **objets** Python, sont associés à des **méthodes**. Le tableau ci-dessous présente quelques méthodes associées aux listes (voir la liste exhaustive sur le Web à l'adresse : *http://docs.python.org/lib/typesseq-mutable.html*)

Méthode	Signification
count(elt)	retourne le nombre d'éléments de la liste qui ont la valeur de *elt*
index(elt)	retourne le plus petit index d'un élément qui a la valeur de *elt*. S'il n'y a aucun élément, déclenche une erreur.
insert(i,elt)	Insert dans la liste avant l'élément à la position *i* la valeur de *elt*
remove(elt)	supprime le premier élément qui a la valeur de elt
reverse()	renverse l'ordre de la liste
sort()	Ordonne la liste selon l'ordre par défaut de ses éléments

3.3 - Les listes

```
>>>texte = u'Le médecin a examiné le malade mais le malade n'était pas vraiment malade.'
>>> liste_mots = texte.split()
>>> lexique = [u'médecin', u'malade']
>>> for mot in lexique :
        i =lexique.index(mot)
        lexique[i] = [mot, liste_mots.count(mot)]

>>> print lexique

[[u'm\xe9decin', 1], [u'malade', 2]]
```

3.3 Listes

1. Ecrire un programme Python pour chacune des spécifications ci-dessous utilisant un minimum de fonctions prédéfinies.

a) Transformer une liste de mots en une chaîne de caractères où les mots sont séparés par un caractère d'espacement.

b) Créer à partir d'une liste donnée une nouvelle liste avec les mêmes éléments dans l'ordre inverse.

c) Ordonner une liste de mots par ordre alphabétique en supprimant les doublons.

d) Dans une chaîne de caractères, détecter toutes les occurrences d'une sous-chaîne donnée. Le résultat est une liste de toutes les positions possibles du début de la sous-chaîne.

3.4 - Les tuples

- Les tuples sont des suites d'objets Python qui peuvent être hétéroclites. Leur type est **tuple** et leur syntaxe est la suivante : $o_1, o_2, \ldots, o_n$ ou $(o_1, o_2, \ldots, o_n)$. Le tuple vide est représenté par *()* .

- Comme les chaînes de caractères et les listes, les tuples sont des objets complexes du type **sequence**. On peut donc leur appliquer les opérateurs + et *. De même, on peut accéder à un élément d'un tuple, ainsi qu'à un sous-tuple par l'opération de tranchage.

- La différence avec les listes est que les tuples ne sont **pas modifiables**. On les utilise à la place des listes quand on ne veut pas que les données soient modifiées par erreur et pour économiser l'espace de stockage.

3.4 - Les tuples

```
>>>tuple = (23, ‘bonjour’, [1,2])
>>>print tuple[2]
[1,2]
>>>tuple[1] = ‘bonsoir’
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in
      <module>
    tuple[1] = 'bonsoir'
TypeError: 'tuple' object does not
      support item assignment
>>>print tuple[1:-1]
( ‘bonjour’,)
>>> tuple += 3,4
>>>print tuple
(23, ‘bonjour’, [1,2], 3, 4)
>>> u,v,x,y,z = tuple
>>> print v
‘bonjour’
```

```
>>>texte = u’Je suis tombé malade. Je n’ai pas travaillé.’
>>> liste_mots = texte.split()
>>> liste_mots_etiquetes = []
>>> for mot in liste_mots :
      liste_mots_etiquetes += [(mot, len(mot))]

>>> print liste_mots_etiquetes
[(u'Je', 2), (u'suis', 4), (u'tomb\xe9', 5), (u'malade.', 7), (u'Je', 2),
      (u'n\u2019ai', 4), (u'pas', 3), (u'travaill\xe9.', 10)]
```

3.5 - Les dictionnaires

- Les dictionnaires sont des collections d'objets Python qui peuvent être hétéroclites et qui sont accessibles à l'aide d'une **clé**. Leur type est **dict** et il est un cas particulier d'un type plus général, le type **mapping**. Leur syntaxe est la suivante : $\{c_1 : o_1, c_2 : o_2, \ldots, c_n : o_n\}$

- On accède à la valeur d'un élément d'un dictionnaire *d* associé à la clé *c*, appelé **entrée**, à l'aide de l'expression *d[c]*.

3.5 - Les dictionnaires

- Les dictionnaires sont des objets **modifiables**. Pour ajouter l'entrée *v* associée à la clé *c* dans le dictionnaire *d*, ou pour la mettre à jour, on utilise l'instruction *d[c] = v* . Pour supprimer l'entrée associée à la clé *c* dans le dictionnaire *d*, on utilise l'appel de fonction *del(d[c]).*

- Comme objets Python, les dictionnaires sont associés à des méthodes dont on peut trouver la liste exhaustive sur le Web à l'adresse : *http://docs.python.org/lib/typesmapping.html*

3.5 - Les dictionnaires

```
>>>d = {‘a’ : 23, 2 : ‘bon’}
>>>print d[‘a’]
23
>>>d[‘b’] = ‘mal’ ; d[‘a’] = 12
>>>print d
{‘a’ : 12, 2 : ‘bon’, ‘b’ : ‘mal’}
>>> del(d[‘a’])
>>>print d
{2 : ‘bon’, ‘b’ : ‘mal’}
>>>
```

```
>>>texte = u’Jean le voit avec le patron’
>>> liste_mots = texte.split()
>>> freq = {}
>>> for mot in liste_mots :
    if freq.has_key(mot):
        freq[mot] += 1
    else:
        freq[mot] = 1

>>> print freq
{u’avec’ :  1, u’Jean’ : 1, u’le’ : 2, u’voit’ : 1, u’patron.', 2}
```

3.5 Les dictionnaires

1. Ecrire un programme Python pour chacune des spécifications ci-dessous.

   a) Compter le nombre d'occurrences des différents caractères dans un texte en utilisant un dictionnaire.

   b) Mémoriser dans un dictionnaire la position de chacun des mots d'un texte sous forme d'un couple (position du début, position de la fin). Si un mot se retrouve plusieurs fois, toutes ses positions doivent être mémorisées.

   c) Inverser un dictionnaire franco-anglais.

   d) A partir d'un texte annoté en parties du discours, constituer un dictionnaire qui donne pour chaque partie du discours, la partie du discours la plus fréquente qui précède et qui suit immédiatement ses occurences dans le texte.