

RÉVISIONS 2 : LES TYPES EN PYTHON

Sommaire

1	Le type chaîne de caractère	3
1.1	Principe d'écriture et affectation	3
1.2	L'encodage utf-8	3
1.3	Opérations de base sur les chaînes de caractères	4
	a - Longueur	4
	b - Parcours des caractères	4
	c - Concaténation - Répétition	5
	d - Conversion (très utile dès maintenant!!!)	5
2	Les listes	6
2.1	Création d'une liste	6
2.2	Interrogation d'une liste - sens d'indexage	7
2.3	Opérations de base sur les listes	7
	a - Parcours et vérification d'une liste	7
	b - Concaténation - Répétition	8
	c - Ajout - suppression	8
	d - Duplication - problème de pointeur mémoire	9
3	Les tuples	9
3.1	Création d'un tuple	9
3.2	Usage courant des tuples	10
4	Les tableaux avec <i>numpy</i>	11
4.1	Intérêt du module <i>numpy</i>	11
4.2	Définition des vecteurs et matrices	11
	a - Par implémentation "directe"	11
	b - Par construction "séquentielle"	12
4.3	Opérations de base sur les tableaux	13
	a - Parcours des éléments - modification	13
	b - Opérations algébriques de base sur les tableaux	14
	c - Quelques opérations d'algèbre linéaire - module bibliothèque <i>linalg</i>	14
5	Les dictionnaires	15

5.1	Structure, création, et balayage des dictionnaires	15
5.2	Opérations sur les dictionnaires	15
6	La bibliothèques PANDAS : Series et DataFrames	17
6.1	Intérêt de Pandas	17
6.2	Création et consultation	17
	a - Cas des Series	17
	b - Cas des DataFrames	19

1 Le type chaîne de caractère

1.1 Principe d'écriture et affectation

Une chaîne de caractères est une suite indicée de caractères et donc **parcourable**, par exemple à l'aide d'une boucle.

Elle est délimitée par des apostrophes ou des guillemets :

```
>>> a='Hello World'
>>> print(a)
Hello World
>>>
```

NB : contrairement aux autres variables en python les chaînes sont **immuables**. Nous reviendrons dessus un peu plus tard.

CARACTÈRES SPÉCIAUX : on peut faire appel à certains caractères spéciaux dans les chaînes à l'aide de la commande (backslash)

```
>>> a='C\'est un beau roman'
>>> print(a) # \' permet d'insérer une apostrophe sans confusion avec un délimiteur de chaîne
C'est un beau roman
>>> a='C\'est un beau roman\n C\'est une belle histoire'
>>> print(a) # \n permet d'insérer une saut de ligne
C'est un beau roman
C'est une belle histoire
>>> a='C\'est un beau roman\t C\'est une belle histoire' # \t permet d'insérer une tabulation
>>> print(a)
C'est un beau roman      C'est une belle histoire
>>>
```

1.2 L'encodage utf-8

On appelle codage utf-8 pour *Universal Character Set Transformation Format - 8 bits* un format universel de codage des caractères. Il permet non seulement un affichage correct des caractères accentués et ce dans toutes les langues, mais également celui d'autres types de caractères standardisés. Pour déclarer une chaîne de caractère dans ce format, on la fait précéder du caractère **u** :

```
>>> a1="célèbre"
>>> a2=u"célèbre"
>>> type(a1);type(a2)
<type 'str'>
<type 'unicode'>
>>>
```

Les conversions du type 'str' en format utf-8 sont réalisés par la méthode **decode** :

```
>>> a1="célèbre"
>>> a2=a1.decode('utf-8')
>>> type(a1);type(a2)
<type 'str'>
<type 'unicode'>
>>>
```

L'opération inverse ('utf-8->'str') se font de la même manière avec la méthode `encode`.

1.3 Opérations de base sur les chaînes de caractères

a - Longueur

La longueur totale d'une chaîne de caractères est obtenue par la commande `len`

```
>>> a1="Papa est en haut"
>>> len(a1)
16
>>>
```

NB : les espaces sont codés et compte donc au même titre que n'importe quel autre caractère.

b - Parcours des caractères

On peut parcourir une chaîne et accéder à n'importe quel caractère de celle-ci à l'aide de son indiciation ; mais on ne peut remplacer un caractère!!!! :

```
>>> a1="papa est en haut"
>>> print a1[0]
p
>>> print a1[len(a1)-1] # attention toujours au premier indice qui vaut 0!!!
t
>>> a1[0]="P" #On tente de mettre une majuscule au début mais....
Traceback (most recent call last) : File "<stdin>", line 1, in <module> TypeError : 'str' object does not support
item assignment
>>>
```

EXERCICE N°1: Réaliser un programme qui extrait chaque mot de la chaîne précédente et les renvoie à l'écran (sans les espaces).

RÉPONSE :

Listing II.1 – Découpage d'une phrase

```
1 a1="Papa_est_en_haut"
2 mot=""
3 N=0
4 while N<=(len(a1)-1):
5     if a1[N]<>"_" and a1[N]<>".": #Détection si changement de mot ou point en fin de phrase
6         mot=mot+a1[N]
7         if N==len(a1)-1: #on est alors au dernier caractère du dernier mot (utile si le
            point a été oublié.)
8             print mot #affiche le dernier mot
9             N=N+1
10        else: #on est à la fin d'un mot
11            print mot #affiche ce mot
12            mot=""
13            N=N+1
```

On peut facilement extraire une sous-chaine à l'aide de l'indiciation et de la syntaxe :

<chaîne>[indice début :indice fin :pas (default=1)]

NB : l'indice fin n'est pas pris en compte :

```
>>> a2="Maman est en bas"
>>> print a2[:4]
Mama
>>> print a2[len(a2)-1:0:-1]
sab ne tse nama #pas de premier caractère!!!
>>>
```

c - Concaténation - Répétition

La concaténation des chaînes est réalisée simplement avec l'opérateur **+**. Lors de cette opération, tous les éléments doivent être de même type, chaîne ici :

```
>>> a1="papa est en haut"
>>> a2="maman est en bas"
>>> afinal=a1+" "+"et"+" "+a2
>>> print afinal
papa est en haut et maman est en bas
>>> afinal*2
papa est en haut et maman est en baspapa est en haut et maman est en bas
>>>
```

d - Conversion (très utile dès maintenant !!!)

On peut convertir toute valeur numérique entière ou en virgule flottante en chaîne de caractères avec **str**. Inversement : toute chaîne représentant une valeur numérique peut-être convertie en entier ou flottant à l'aide des commandes respectives **int** et **float** :

```

>>> a1="Monsieur python habite au n°"
>>> a2=str(123)
>>> type(a2)
<type 'str'>
>>> print a1+a2
Monsieur python habite au n°123
>>> a2=int(a2)
>>> type(a2)
<type 'int'>
>>> a2=str(a2)
>>> type(a2)
<type 'str'>
>>> a2=float(a2)
>>> type(a2)
<type 'float'>
>>>

```

2 Les listes

2.1 Création d'une liste

Nous avons déjà rencontré la notion de **liste** à l'occasion des rappels consacrés à la commande de boucle *for*. Nous devons fournir une liste déjà constituée antérieurement, ou bien créée au sein de la boucle par la commande **range**.

```

>>> liste1=[1,2,3,4]
>>> type(liste1)
<type 'list'>
>>> liste1
[1, 2, 3, 4]
>>> liste2=range(1,5,1)
[1, 2, 3, 4]
>>>

```

Une autre méthode de création de liste est appelée "méthode par compréhension". On utilise simplement une boucle :

```

>>> liste=[ 2*i+1 for i in range(10)]
>>> liste
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>>

```

NB : une autre syntaxe de la commande *range* permet de spécifier les premiers et derniers indices, ainsi que le pas :

```

>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>>

```

Mais une liste peut également comporter des éléments non numériques, par exemple des chaînes de caractères, et même des listes comme éléments :

```
>>> liste3=["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche",[1,2,3]]
>>> liste3
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi', 'dimanche', [1, 2, 3]]
>>>
```

2.2 Interrogation d'une liste - sens d'indiciage

Chaque élément d'une liste est identifié par un index. Le premier terme de la liste est identifié par l'indice 0 :

```
>>> liste=["petit","moyen","grand","immense"]
>>> print(liste[0],liste[3])
('petit', 'immense')
>>>
```

Une liste peut également être indexée avec des nombres négatifs et l'on peut interroger ses éléments selon le modèle suivant :

```
>>> print(liste[-1],liste[-2],liste[-3],liste[-4])
('immense', 'grand', 'moyen', 'petit')
>>>
```

NB : l'intérêt est de pouvoir interroger le dernier élément de la liste sans connaître le nombre d'éléments de celle-ci. On peut également extraire des sous-liste d'une liste :

```
>>> liste=[1, 2, [3, 4], "cinq", 6]
>>> liste[:3]      # les 3 premiers
[1, 2, [3, 4]]
>>> print liste[1 : 2]    # de 2 en 2 à partir du deuxième
[2, 'cinq']
>>> print liste[1 :]      # tout à partir du deuxième
[2, [3, 4], 'cinq', 6]
>>> print liste[-3 :]     # les 3 derniers
[[3, 4], 'cinq', 6]
>>> print liste[3 :0 :-1]  # du 4ième au 2ième en sens inverse
['cinq', [3, 4], 2]
```

2.3 Opérations de base sur les listes

a - Parcours et vérification d'une liste

On peut parcourir les éléments d'une liste à l'aide d'une boucle inconditionnelle *for* et de l'instruction **in** :

```
>>> liste=[1,2,[3,4],"cinq",6]
>>> for elt in liste : print(elt)
1
2
[3, 4]
cinq
6
```

in permet également de déterminer la présence d'un élément dans une liste sans réaliser une boucle de parcours complète (ce qui est une autre méthode plus lourde) :

```
>>> if "cinq" in liste : print("présent")
présent
>>>
```

NB : l'instruction **len** permet de connaître la longueur d'une liste :

```
>>> len(liste)
5
>>>
```

b - Concaténation - Répétition

Les listes sont concaténées et répétées à l'aide des opérateurs **+** et ***** (comme les chaînes) :

```
>>> liste1=[1,2,3]
>>> liste2=[4,5,6]
>>> liste3=liste1+liste2
>>> print (liste3)
[1, 2, 3, 4, 5 6]
>>> liste4=liste1*3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

c - Ajout - suppression

On peut réaliser l'insertion de d'éléments dans une liste à l'aide des méthodes **append(<objet>)**, pour une insertion en fin de liste, et **insert(<indice>,<objet>)** pour une insertion à la position **<indice>** :

```
>>> liste=[1,2,3,5,6]
>>> liste.insert(3,4)
>>> print(liste)
[1, 2, 3, 4, 5, 6]
>>> liste.append(7)
>>> print(liste)
[1, 2, 3, 4, 5, 6, 7]
>>>
```

On peut également insérer une sous-liste ainsi :

```
>>> liste=[1,2,5,6]
>>> liste[2:2]=[3,4]
>>> print(liste)
[1, 2, 3, 4, 5, 6]
>>>
```

La suppression d'éléments de liste se fait à l'aide de la commande **del** :


```
>>> liste=[1, 2, 3, 4, 5, 6]
>>> del liste[2:]      # supprime tous les éléments après l'indice 2 compris
>>> print(liste)
[1, 2]
>>>
```

Enfin, la méthode **remove** permet d'extraire la première occurrence d'un élément dans une liste :

```
>>> liste=[1, 2, 3]*3
>>> print(liste)
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> liste.remove(2)
>>> print(liste)
[1, 3, 1, 2, 3, 1, 2, 3]
>>>
```

d - Duplication - problème de pointeur mémoire

Lorsque l'on tente de dupliquer une liste en recopiant celle-ci sous un autre nom, on ne fait que créer une nouvelle étiquette qui pointe vers la même adresse mémoire. Ainsi, la modification d'un élément de liste en exploitant l'une des deux étiquettes vaut pour les deux étiquettes. On ne réalise donc pas de véritable duplication :

```
>>> L1=[1,2,3,4,5]
>>> L2=L1
>>> L1[2]=3.5
>>> print(L2)
[1, 2, 3.5, 4, 5]
>>>
```

On exploite pour la duplication la commande **deepcopy** du module **copy** :

```
>>> L1=[1,2,3,4,5]
>>> from copy import deepcopy
>>> L2=deepcopy(L1)
>>> L1[2]=3.5
>>> print(L1,L2)
([1, 2, 3.5, 4, 5], [1, 2, 3, 4, 5])
>>>
```

3 Les tuples

3.1 Création d'un tuple

Un tuple est similaire à une liste à deux différences près :

- ses délimiteurs sont des parenthèses et non des crochets
- il est **immuable** (ou non mutable) comme les chaînes de caractères

```
>>> t=(1,2,3,4)
>>> type(t)
<type 'tuple'>
>>> t[0]=0
Traceback (most recent call last) : File "<stdin>", line 1, in <module> TypeError : 'tuple' object does not support
item assignment
>>>
```

On peut également exploiter la fonction `zip(L1,L2)` qui permet de créer une liste de tuples à partir de deux listes ($L1, L2$) :

```
>>> L1=["anaïs","anatole","benoit","caroline","laurent","rebecca","sandra","xavier"]
>>> L2=["20","29","28","42","38","15","34","52"]
>>> listeage=zip(L1,L2)
>>> print listeage
[('anaïs', '20'), ('anatole', '29'), ('benoit', '28'), ('caroline', '42'),
 ('laurent', '38'), ('rebecca', '15'), ('sandra', '34'), ('xavier', '52')]
```

Si les listes à "tupler" ne sont pas de même longueur, alors la commande `zip(L1,L2)` fabrique les tuples en prenant les éléments de même indice dans chaque liste ; les éléments surnuméraires de la plus longue des deux listes seront ignorés lors de la fusion :

```
>>> L1=L1+["zebulon"]
>>> listeage=zip(L1,L2)
>>> print listeage
[('anaïs', '20'), ('anatole', '29'), ('benoit', '28'), ('caroline', '42'),
 ('laurent', '38'), ('rebecca', '15'), ('sandra', '34'), ('xavier', '52')]
```

3.2 Usage courant des tuples

On peut exploiter les tuples dans les boucles inconditionnelles *for* :

```
>>> for (i,j) in [(0,'a'),(1,'b'),(2,'c'),(3,'d')] :print i,j
0 a
1 b
2 c
3 d
>>> for (i,j) in enumerate(['a','b','c','d']) :print i,j
0 a
1 b
2 c
3 d
>>>
```

NB : remarquez la commande `enumerate` qui permet de constituer des tuples (i,j) avec i qui prendra la valeur de l'indice d'un élément et j la valeur de l'élément en question.

4 Les tableaux avec *numpy*

4.1 Intérêt du module *numpy*

Nous avons déjà évoqué les listes en Python qui sont en fait des **tableaux** dont le contenu est hétérogène et dont la profondeur peut être modifiée très simplement.

Le module **numpy** de Python permet de manipuler de manière plus performante les tableaux, avec entre autres avantages :

- La possibilité de définir des tableaux multidimensionnels (et pas simplement des listes de listes) avec des éléments homogènes.
- une implémentation rigoureuse en mémoire, donc proche du hardware permettant **un important gain de temps dans les calculs (opérations matricielles par exemple)**.
- un accès à toutes les fonctions mathématiques classiques (**sin, cos, e, pi, etc...**) et **applicables par exemple aux tableaux élément par élément**.

4.2 Définition des vecteurs et matrices

a - Par implémentation "directe"

On peut déclarer un tableau et l'implémenter en mémoire en écrivant directement l'ensemble de ses éléments : on utilise la commande `numpy.array([[ligne 1],[ligne 2],[ligne 3],...,[ligne n]])`

Diverses méthodes : `.ndim`, `.shape`, `.dtype` permettent d'extraire des informations concernant un tableau ou bien d'en modifier la structure. On donne quelques exemples de manipulations ci-dessous.

NB : on charge d'abord le module *numpy*.

```
>>> import numpy as np # permet d'avoir un alias plus court à écrire
>>> tab=np.array([[1,2,3],[4,5,6]])
>>> print(tab)
[[1, 2, 3], [4, 5, 6]]
>>> tab.ndim
2 #tableau à deux dimensions
>>> tab.shape # interroge sur la taille ligne(s) et colonne(s)
(2, 3)
>>> tab.dtype #interroge sur le type des éléments du tableau
dtype('int32') # que des entiers!!!
>>> tab=np.array([[1,2,3],[4,5,6.1]])
>>> tab.dtype
dtype('float64') # et maintenant que des flottants car homogénéité requise!!!
>>> tab.shape=(3,2) #modifie nombres de ligne(s) et colonne(s)
>>> tab
array([[ 1. ,  2. ], [ 3. ,  4. ], [ 5. ,  6.1]]) # et maintenant 3 lignes et deux colonnes
```

On peut également définir des tableaux unité ou nuls :

```

>>> tab=np.zeros((4,3)) # attention de ne pas oublier le second jeux de parenthèses autour du tuple de taille
>>> tab
array([[ 0.,  0.,  0.], [ 0.,  0.,  0.], [ 0.,  0.,  0.], [ 0.,  0.,  0.]])
>>> tab=np.ones((4,3))
>>> tab
array([[ 1.,  1.,  1.], [ 1.,  1.,  1.], [ 1.,  1.,  1.], [ 1.,  1.,  1.]])
>>> vect=np.arange(10) # permet de construire un tableau 1D facilement
>>> vect
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> vect.ndim
1
>>> tab=np.identity(3) # crée une matrice identité de dimension 3*3
>>> tab
array([[ 1.,  0.,  0.], [ 0.,  1.,  0.], [ 0.,  0.,  1.]])

```

b - Par construction "séquentielle"

A la manière des listes que l'on peut former avec la commande `range` il est possible de construire un tableau 1D d'éléments régulièrement espacés en utilisant la commande `arange` :

```

>>> from scipy import *
>>> arange(15)
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
>>> arange(2, 3, 0.1)
array([ 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])

```

ou encore un tableau 1D d'éléments comportant un nombre spécifié d'éléments régulièrement espacés :

```

>>> from scipy import *
>>> linspace(-1., 4., 11)
array([-1. , -0.5,  0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ])
>>> logspace(-1, 4, 11)
array([ 1.00000000e-01, 3.16227766e-01, 1.00000000e+00, 3.16227766e+00, 1.00000000e+01, 3.16227766e+01,
1.00000000e+02, 3.16227766e+02, 1.00000000e+03, 3.16227766e+03, 1.00000000e+04])

```

Enfin, il est possible d'employer la commande `numpy.append` (différent de la méthode `append` employée pour les listes) afin de construire des tableaux (matrices) ligne par ligne, ou bien colonne par colonne à partir de listes. La syntaxe est la suivante :

```

>>> import numpy as np
>>> tab=np.array(np.zeros((1,4)),dtype=int)
>>> tab
array([[0, 0, 0, 0]])
>>> Ltab=[[1,1,1,1]]
>>> tab=np.append(tab,Ltab,axis=0) # axis=0 permet l'ajout de Ltab à la matrice tab en tant que ligne
>>> tab
array([[ 0,  0,  0,  0], [ 1,  1,  1,  1]])
>>> Ctab=[1],[1]
>>> tab=np.append(tab,Ctab,axis=1) # axis=1 permet l'ajout de Ctab à la matrice tab en tant que colonne
>>> tab
array([[0, 0, 0, 0, 1],[1, 1, 1, 1, 1]])

```

4.3 Opérations de base sur les tableaux

a - Parcours des éléments - modification

L'appel d'un élément de tableau se fait selon l'ordre habituel d'indiciation. Par exemple pour le tableau **tab**, l'élément de la $i^{\text{ième}}$ ligne et $j^{\text{ième}}$ colonne est appelé par :

`tab[i,j]` ou bien `tab[i][j]` ou encore avec un tuple `tab[(i,j)]`

EXERCICE N°2: *Que fait le script suivant :*

Listing II.2 – Que fais-je ?

```
1 import numpy as np
2 tab=np.array ([[1,2],[3,4],[5,6]])
3 for l in range(tab.shape[0]):
4     print tab[l,:]
5 for c in range(tab.shape[1]):
6     print tab[:,c]
```

On peut également modifier n'importe quel élément puisque **les tableaux sont mutables**. On peut également modifier la dimensionnalité d'un tableau à l'aide de la méthode **reshape**. L'extrait de console ci-dessous illustre cela :

```
>>> import numpy as np
>>> tab=np.arange(4)
>>> tab
array([0, 1, 2, 3])
>>> tab.ndim
1
>>> tab.shape
(4,)
>>> tab2=tab.reshape(4,1)
>>> tab2
array([[0],
       [1],
       [2],
       [3]])
>>> tab2.ndim
2
>>> tab2.shape
(4,1)
```

Enfin, il est possible d'extraire n'importe quelle sous-liste d'une liste par la technique de **slicing** (littéralement découpage par "tranches") :

```
>>> tableau1=np.arange(20).reshape(4,5)
>>> tableau1
array([[ 0,  1,  2,  3,  4],[ 5,  6,  7,  8,  9],[10, 11, 12, 13, 14],[15, 16, 17, 18, 19]])
>>> tableau2=tableau1[1:3, 3:]
>>> tableau2
array([[ 8,  9],[13, 14]])
```

La force de **numpy** est de pouvoir réaliser des opérations sur tous les éléments d'un tableau simultanément sans passer par une boucle couteuse en ressources machine. On donne quelques exemples ci-dessous :

numpy permet également d'appliquer une fonction mathématique à tous les éléments d'une matrice. Ainsi, on a par exemple sur un tableau de dimension 1 :

c - Quelques opérations d'algèbre linéaire - module bibliothèque linalg

- inversion d'une matrice

```
>>> tableau=np.array([1,3,5,7,9],float).reshape(2,2)
>>> np.linalg.inv(tableau)
array([[ -0.875,  0.375],[ 0.625, -0.125]])
```

- résolution d'un système linéaire type $A \cdot X = B$

```
>>> A=np.array([1,3,5,7],float).reshape(2,2)
>>> B=np.array([2,5],float)
>>> np.linalg.solve(A,B)
array([ 0.125, 0.625])
```

- diagonalisation de matrice \Rightarrow à voir plus tard en fonction de l'avancement du cours de mathématiques.

5 Les dictionnaires

5.1 Structure, création, et balayage des dictionnaires

Il s'agit d'un ensemble non ordonné de couples de forme :

`<clé>:<valeur>`

les délimiteurs du dictionnaire étant les accolades (`{.....}`).

La recherche d'une valeur dans un dictionnaire se fait à l'aide de sa **clé**. Les clés peuvent être de tout type, **hormis une liste**. Les valeurs peuvent en revanche être rigoureusement de tout type.

On crée et interroge un dictionnaire de la façon suivante :

```
>>> dictioAF={} #le dictionnaire est vide
>>> dictioAF=dictioAF+{"screen":u"écran","table": "table","desk": "bureau", "watch": "montre"}
Traceback (most recent call last) :
File "<stdin>", line 1, in <module>
TypeError : unsupported operand type(s) for + : 'dict' and 'dict'
>>> dictioAF={"screen":u"écran","table": "table","desk": "bureau", "watch": "montre"}
>>> print dictioAF("desk") #interrogation d'un élément par sa clé
bureau
>>> print dictioAF["computer"] #interrogation par une clé non présente dans le dictionnaire
Traceback (most recent call last) :
File "<stdin>", line 1, in <module>
KeyError : 'computer'
>>> print dictioAF.get("computer","ordinateur") #si clé non présente, on peut renvoyer une valeur alternative
>>> print dictioAF.get("desk","ordinateur") #si clé présente, la valeur alternative est ignorée
bureau
```

5.2 Opérations sur les dictionnaires

- Ajout et suppression d'élément

Les dictionnaires sont mutables, ainsi :

```
>>> dictioAF["sofa"]="canapé"
>>> dictioAF
{'table' : 'table', 'screen' : u'écran', 'sofa' : u'canapé', 'watch' : 'montre', 'desk' : 'bureau'}
>>> dictioAF["sofa"]="divan"
>>> dictioAF
{'table' : 'table', 'screen' : u'écran', 'sofa' : 'divan', 'watch' : 'montre', 'desk' : 'bureau'}
```

On peut faire appel à l'instruction `del` ou à la méthode `.pop(<clé>)` pour supprimer un élément par l'intermédiaire de sa clé :

```
>>> del dictioAF["screen"] >>> dictioAF
{'table' : 'table', 'sofa' : 'divan', 'watch' : 'montre', 'desk' : 'bureau'}
>>> dictioAF.pop("table") 'table' #attention : .pop supprime mais renvoie la valeur
>>> dictioAF
{'sofa' : 'divan', 'watch' : 'montre', 'desk' : 'bureau'}
```

- **Présence et énumération des éléments**

On peut tester la présence d'un élément dans la liste à partir de sa clé par la méthode `.has_key(<clé>)` :

```
>>> dictioAF.has_key("screen")
False >>> dictioAF.has_key("sofa")
True
```

On peut enfin énumérer le contenu d'un dictionnaire, ou seulement ses clés, ou encore ses valeurs par les méthodes respectives `.items()` et `.keys()`, `.values()` :

```
>>> dictioAF.items()
[('sofa', 'divan'), ('watch', 'montre'), ('desk', 'bureau')]
>>> dictioAF.keys()
['sofa', 'watch', 'desk']
>>> dictioAF.values()
["divan", "montre", "bureau"]
```

- **Parcours d'un dictionnaire**

Enfin, il est possible de parcourir un dictionnaire à l'aide d'une boucle `for`, l'itération se faisant sur les clés :

```
>>> for cle in dictioAF : print u"clé,valeur : ",cle,dictioAF[cle]
clé,valeur : sofa divan
clé,valeur : watch montre
clé,valeur : desk bureau
```


6 La bibliothèques PANDAS : Series et DataFrames

6.1 Intérêt de Pandas

La bibliothèque **Pandas** permet de créer et manipuler des données dans des «contenants» appelées **Series** (assez similaires à des tableaux 1D, voire des vecteurs) ou **DataFrames** (assez similaires à des tableaux). L'immense intérêt réside dans la facilité de manipulation de ces types, en particulier leur création à partir d'un fichier externe tabulé (par exemple un fichier Excel® **.xls** ou bien plus généralement **.csv** ou **.text**). En outre, le problème bien connu de la gestion des séparateurs de données ("," ou bien ";") dans les fichiers **.csv** ou **.txt**, imposant l'emploi d'une commande **.split** pour une importation sous forme d'un tableau dans Python n'existe tout simplement plus avec **Pandas**.

Enfin, les **DataFrames** présentent également l'avantage de pouvoir contenir des étiquettes descriptives pour les lignes et colonnes, exactement comme dans un tableur, permettant alors une bien meilleure lisibilité et une plus grande convivialité.

L'extraction de données de ces **DataFrames** est en outre aussi simple que dans le cas des tableaux classiques, à la manière du slicing des tableaux, par exemple pour réaliser des graphiques avec le module **matplotlib**.

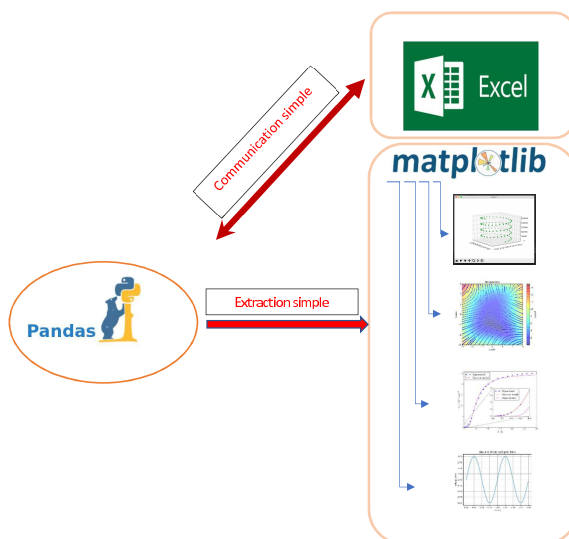


FIGURE II.1 – Communication du module **Pandas** avec l'extérieur.

Pandas, nécessaire à la manipulation de ce type de données, doit être importé par la commande classique suivante :

```
>>> import pandas
```

6.2 Création et consultation

a - Cas des Series

La déclaration d'une variable de type **Series** est assez similaire à celle d'un tableau 1D (**attention au S majuscule de Series**) :

```
>>> s=pandas.Series([1, 2, 5, 7]) pour une série numérique entière
>>> print(s)
0    1
1    2
2    5
3    7
>>> s=pandas.Series([1.3, 2, 5, 7]) même chose mais série numérique flottante cette fois
>>> print(s)
0    1.3
1    2.0
2    5.0
3    7.0
```

On peut également indiquer le type de données stockées dans la série lors de sa déclaration, à la manière de ce que nous faisons habituellement dans les tableaux :

```
>>> s=pandas.Series([1, 2, 5, 7], dtype=numpy.float64) pour une série numérique en flottants codés sur 64 bits
>>> print(s)
0    1.3
1    2.0
2    5.0
3    7.0
```

Ci-dessous les manipulations simples des séries :

On peut interroger la dimension de la series

```
>>>len(s)
```

```
4
```

```
>>>s.size
```

```
4
```

On peut faire facilement une copie de la série

```
>>> s2=s.copy()
```

```
>>>print(s2)
```

```
0    1.3
```

```
1     2.0
```

```
2     5.0
```

```
3     7.0
```

```
>>> s=pandas.Series([1.3, 2, 5, 7], index = ['a', 'b', 'c', 'd']) permet de libeller chaque valeur
```

```
>>> print(s)
```

```
a    1.3
```

```
b     2.0
```

```
c     5.0
```

```
d     7.0
```

On peut interroger des éléments de la série à l'aide des indices, de la même manière qu'avec des tableaux `numpy` :

```
>>> s[2]
```

```
5.0
```

ou bien à l'aide des libellés avec les méthodes `.at`, `.loc`, `.get` :

```
>>> s['c']
```

```
5.0
```

```
>>> s.at['c']
```

```
5.0
```

```
>>> s.loc['c']
```

```
5.0
```

```
>>> s.get['c']
```

```
5.0
```

Comme pour les tableaux, le slicing est très utile :

```
>>> s[2 :]
```

```
c    5.0
```

```
d    7.0
```

```
>>> s[0 :2]
```

```
a    1.3
```

```
b     2.0
```

On peut convertir en tableau `numpy` :

```
>>>stab=s.values
```

```
>>>print(stab)
```

```
[1.3 2. 5. 7.]
```

```
>>>type(stab)
```

```
numpy.ndarray
```

b - Cas des DataFrames

Un **Dataframe** se comporte comme un tableau `numpy` dont les colonnes sont identifiées par des clés à la manière d'un dictionnaire, et les valeurs en colonne des **Series**. On peut créer un **DataFrame** à l'aide d'un tableau :

```

>>> tab=numpy.array([[1.1, 2, 3.3, 4], [2.7, 10, 5.4, 7], [5.3, 9, 1.5, 15]])
>>> df = pandas.DataFrame(tab)
>>> print(df)
   0    1    2    3
0  1.1  2.0  3.3  4.0
1  2.7 10.0  5.4  7.0
2  5.3  9.0  1.5 15.0

```

On peut également déclarer des noms de colonnes et de lignes :

```

>>> df1 = pandas.DataFrame(tab, index = ['a1', 'a2', 'a3'], columns = ['A', 'B', 'C', 'D'])
>>> print(df1)
   A    B    C    D
a1  1.1  2   3.3  4.0
a2  2.7 10   5.4  7.0
a3  5.3  9   1.5 15.0

```

On peut également créer un Dataframe en détaillant chaque colonne (noter l'inversion des colonnes 'B' et 'C') :

```

>>> df2 = pandas.DataFrame('A' : [1.1, 2.7, 5.3], 'B' : [2, 10, 9], 'C' : [3.3, 5.4, 1.5], 'D' : [4, 7, 15], columns =
['A', 'C', 'B', 'D'])
>>> print(df2)
>>> print(df2)
   A    C    B    D
0  1.1  3.3  2   4.0
1  2.7  5.4 10   7.0
2  5.3  1.5  9  15.0

```

La consultation des `DataFrames` peut se faire à l'aide du slicing ou bien des méthodes `.loc` et `.iloc` suivant ce que l'on cherche à faire :

```
>>> df1['A'] pour extraire la Series de la colonne 'A'
a1  1.1
a2  2.7
a3  5.3
>>> df1[0 :2]['A'] pour extraire les deux premières lignes de la colonne 'A'
A
a1  1.1
a2  2.7
>>> df1.loc['a2'] renvoie la Series de la ligne d'index 'a2'
A    2.7
B   10.0
C    5.4
D    7.0
>>> df1.loc[:,['A', 'C']] pour extraire toutes les lignes des colonnes 'A' et 'C'
A    C
a1  1.1  3.3
a2  2.7  5.4
a3  5.3  1.5
>>> df1.loc['a2', 'C'] permet d'accéder à la valeur de la ligne 'a2' colonne 'C'
5.4
>>> df1.loc[3]=[1.7,5.6,4.2,9.4] ajoute une ligne à df1
>>> df1
   A    B    C    D
a1  1.1  2   3.3  4.0
a2  2.7  10  5.4  7.0
a3  5.3  9   1.5  15.0
3   1.7  5.6  4.2  9.4
Enfin, la méthode i.loc permet d'accéder à tous les éléments par indices :
>>> df1.iloc[1] renvoie la seconde ligne
A    2.7
B   10.0
C    5.4
D    7.0
>>> df1.iloc[:,0 :2] : renvoie toutes les lignes des colonnes 0 à 2 exclue
   A    B
a1  1.1  2.0
a2  2.7  10.0
a3  5.3  9.0
3   1.7  5.6
>>> df1.iloc[1,2] : renvoie la valeur à la ligne 2 et la colonne 3
5.4
Comme pour les Series, on peut convertir un DataFrame en tableau numpy :
>>> df1.values
array([[ 1.1,  2. ,  3.3,  4. ], [ 2.7, 10. ,  5.4,  7. ], [ 5.3,  9. ,  1.5, 15. ], [ 1.7,  5.6,  4.2,  9.4]])
```

Enfin, on peut renvoyer les dimensions d'un **DataFrame** avec les commandes et méthodes suivantes :

```
>>> df2.shape renvoie un tuple indiquant le nombre de lignes et de colonnes  
(3,4)  
>>> len(df2) renvoie le nombre de lignes  
3  
>>> len(df2.index) idem  
3  
>>> len(df2.columns) renvoie le nombre de colonnes  
4
```