Révisions 2: les types en Python

(Révisions MPSI)

JL GRAYE

MP3 Lycée Montaigne

Plan

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 guillements :

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 guillements :

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

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 guillements :

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

NB : contrairement aux autres variables en python les chaînes sont **immuables ou non mutables**. (cf plus bas)

Principe d'écriture et affectation

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

Principe d'écriture et affectation

CARACTÈRES SPÉCIAUX : on peut faire appel à certains caractères spéciaux dans les chaînes à l'aide de la commande *d'échappement* \ (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
>>>
```

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 chaine de caractère dans ce format, on la fait précéder du caractère **u** :

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 chaine 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'>
>>>
```

L'encodage utf-8

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

L'encodage utf-8

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 fait de la même manière avec la méthode encode.

Opérations de base sur les chaînes de caractères

Longueur

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

Opérations de base sur les chaînes de caractères

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.

Opérations de base sur les chaînes de caractères

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 ! ! ! ! :

Opérations de base sur les chaînes de caractères

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
>>>
```

Opérations de base sur les chaînes de caractères

Parcours des caractères

EXERCICE N°1:

Réaliser un programme qui extrait chaque mot de la chaîne a1="papa est en haut" *et le renvoie à l'écran (***sans les espaces***).*

Opérations de base sur les chaînes de caractères

Parcours des caractères

RÉPONSE :

```
1  a1="Papa␣est␣en␣haut"
2  mot=""
3  N=0
4  while N<=(len(a1)-1):
5      if a1[N]<>"␣" and a1[N]<>".": #DÃ©tecte 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 caractÃ¨re
       du dernier mot (utile si le point a Ã©tÃ© oubliÃ©.)
8              print mot #affiche le dernier mot
9          N=N+1
10     else: #on est Ã  la fin d'un mot
11         print mot  #affiche ce mot
12         mot=""
13         N=N+1
```

Opérations de base sur les chaînes de caractères

Parcours des caractères

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

Opérations de base sur les chaînes de caractères

Parcours des caractères

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

<chaine>[indice début :indice fin :pas (défault=1)]

Attention :

Opérations de base sur les chaînes de caractères

Parcours des caractères

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

<chaine>[indice début :indice fin :pas (défault=1)]

Attention : 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!!!
>>>
```

Opérations de base sur les chaînes de caractères

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 :

Opérations de base sur les chaînes de caractères

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
>>>
```

Opérations de base sur les chaînes de caractères

Conversion

On peut convertir toute valeur numérique entière ou en virgule flottante en chaine de caractères avec str. Inversement : toute chaine 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'>
>>>
```

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 devions 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]
>>>
```

Création d'une liste

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]
>>>
```

Création d'une liste

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 chaines 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]]
>>>
```

Interrogation d'une liste - sens d'indiçage

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

Interrogation d'une liste - sens d'indiçage

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')
>>>
```

Interrogation d'une liste - sens d'indiçage

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

	"petit"	"moyen"	"grand"	"immense"
indices positifs	0	1	2	3
indices négatifs	−4	−3	−2	−1

Interrogation d'une liste - sens d'indiçage

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

	"petit"	"moyen"	"grand"	"immense"
indices positifs	0	1	2	3
indices négatifs	−4	−3	−2	−1

```
>>> 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 connaitre le nombre d'éléments de celle-ci.

Interrogation d'une liste - sens d'indiçage

On peut également extraire des sous-listes 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]
```

Opérations de base sur les listes

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
```

Opérations de base sur les listes

Parcours et vérification d'une liste

La commande 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
>>>
```

L'instruction len permet de connaitre la longueur d'une liste :

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

Opérations de base sur les listes

Concaténation - Répétition

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

```
>>> 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]
```

Opérations de base sur les listes

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]
>>>
```

Opérations de base sur les listes

Ajout - suppression

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]
>>>
```

Opérations de base sur les listes

Ajout - suppression

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 occurence 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]
>>>
```

Opérations de base sur les listes

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]
>>>
```

Opérations de base sur les listes

Duplication - problème de pointeur mémoire

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])
>>>
```

Création d'un tuple

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

Création d'un tuple

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

- ses délimiteurs sont des parenthèses et non des crochets

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

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
>>>
```

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.

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.

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 :

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 possibilités de définir des tableaux multidimensionnels (et pas simplement des listes de listes) avec des éléments homogènes.

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 possibilités 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).**

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 possibilités 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.**

Définition des vecteurs et matrices

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 :

Définition des vecteurs et matrices

Implémentation directe

NB : on charge d'abord le module numpy nécessaire à l'implémentation mémoire des tableaux.

```
>>> 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 2 colonnes
```

Définition des vecteurs et matrices

Implémentation directe

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) # créé une matrice identité de dimension 3*3
>>> tab
array([[ 1., 0., 0.], [ 0., 1., 0.], [ 0., 0., 1.]])
```

Définition des vecteurs et matrices

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])
```

Définition des vecteurs et matrices

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])
```

Définition des vecteurs et matrices

Par construction "séquentielle"

ou encore un tableau 1D d'éléments comportant un nombre spécifié d'élements 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])
```

Définition des vecteurs et matrices

Par construction "séquentielle"

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]])
```

Opérations de base sur les tableaux

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 $i^{ième}$ ligne et $j^{ième}$ 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 :

```
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]
```

Opérations de base sur les tableaux

Parcours des éléments - modification

On peut également modifier n'importe quel élément puisque les tableaux **ne sont pas un type immuable**. 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)
>>>
```

Opérations de base sur les tableaux

Opérations algébriques de base sur les tableaux

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 :

```
>>> import numpy as np
>>> tab1=np.array([ [1,2], [3,4], [5,6] ])
>>> tab2=0.1*tab1
>>> tab2
array([[ 0.1, 0.2], [ 0.3, 0.4], [ 0.5, 0.6]]) # 3 lignes, 2 colonnes
>>>tab3=tab1+tab2
>>> tab3
array([[ 1.1, 2.2], [ 3.3, 4.4], [ 5.5, 6.6]])
>>> tab4=np.array([[1,2,1],[2,1,2]]) # 2 lignes, 3 colonnes
>>> tab5=tab3.dot(tab4) # réalise le produit de deux matrices compatibles
>>> tab5
array([[ 5.5, 4.4, 5.5], [ 12.1, 11. , 12.1], [ 18.7, 17.6, 18.7]])
>>> tab6=tab5.transpose() #calcule la transposée d'une matrice
>>> tab6
array([[ 5.5, 12.1, 18.7], [ 4.4, 11. , 17.6], [ 5.5, 12.1, 18.7]])
```

Opérations de base sur les tableaux

Opérations algébriques de base sur les tableaux

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 :

```
>>> x=np.arange(0,np.pi,0.1) #crée un tableau de valeurs entre 0 et pi par pas de 0.1
>>> print x
array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3. , 3.1])
>>> y1=np.sin(x)
>>> y2=np.cos(x)
>>> y1
array([ 0. , 0.09983342, 0.19866933, 0.29552021, 0.38941834, 0.47942554, 0.56464247, 0.64421769, 0.71735609, 0.78332691, 0.84147098, 0.89120736, 0.93203909, 0.96355819, 0.98544973, 0.99749499, 0.9995736 , 0.99166481, 0.97384763, 0.94630009, 0.90929743, 0.86320937, 0.8084964 , 0.74570521, 0.67546318, 0.59847214, 0.51550137, 0.42737988, 0.33498815, 0.23924933, 0.14112001, 0.04158066])
>>> y1**2+y2**2
array([ 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Opérations de base sur les tableaux

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

RANG D'UNE MATRICE

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

INVERSION D'UNE MATRICE

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

Opérations de base sur les tableaux

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

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 ⇒ À VOIR PLUS TARD EN FONCTION DE L'AVANCEMENT DU COURS DE MATHÉMATIQUES.

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.

Structure, création, et balayage des dictionnaires

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
ordinateur
>>> print dictioAF.get("desk","ordinateur") #si clé présente, la valeur alternative est ignorée
bureau
```

Opérations sur les dictionnaires

Ajout et suppression d'élément

Les dictionnaires sont mutables, ainsi :

```
>>> dictioAF["sofa"]=u"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'}
```

Opérations sur les dictionnaires

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
```

Opérations sur les dictionnaires

On peut aussi 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"]
```

Opérations sur les dictionnaires

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
```

Intérêt de Pandas

La bibliothèque `Pandas` permet :

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)

Intérêt de Pandas

La bibliothèque `Pandas` permet :

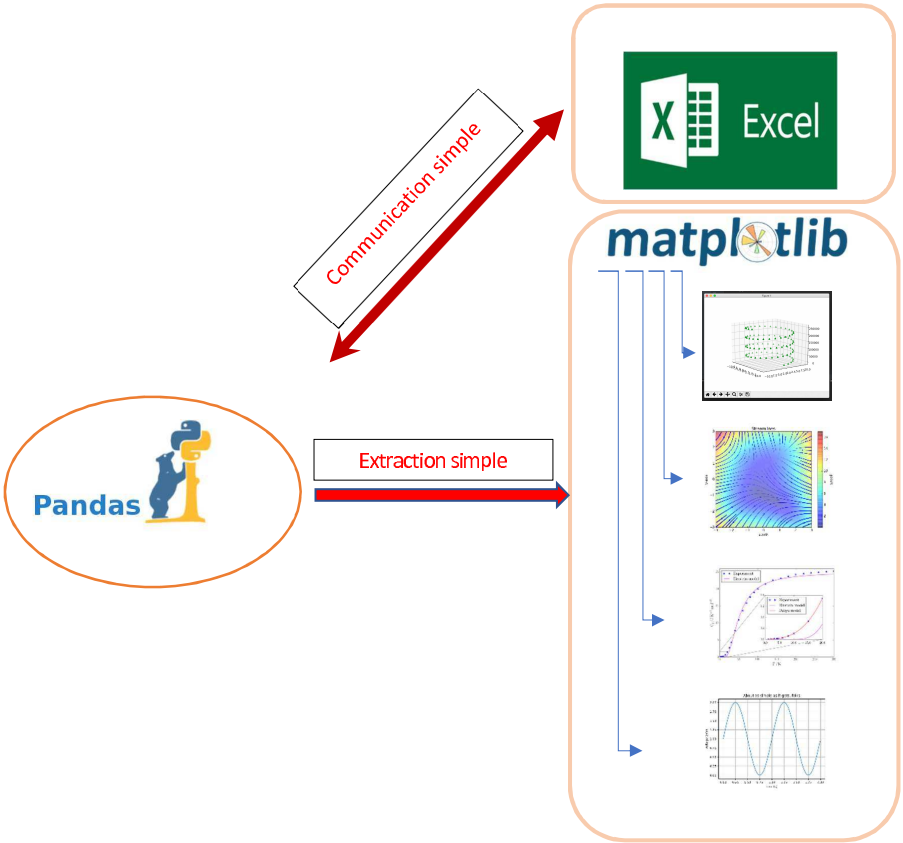
- de créer et manipuler des données dans des «contenants» appelées `Series` (assez similaires à des tableaux 1D, voire des vecteurs) ou `DataFrames` (assez similaires à des tableaux)
- de communiquer très facilement avec les fichiers externes, de format divers (`.csv`, `.txt`, `.xls` etc...)

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)
- de communiquer très facilement avec les fichiers externes, de format divers (`.csv`, `.txt`, `.xls` etc...)
- de s'affranchir facilement des problèmes de séparateurs de données ("," ou " ;") dans la liaison avec des fichiers externes en lecture comme en écriture

Intérêt de Pandas



Intérêt de Pandas

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

```
>>> import pandas
```

Création et consultation

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
```

Création et consultation

Cas des Series

```
>>> 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
```

Création et consultation

Cas des Series

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
```

Création et consultation

Cas des Series

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 copy de la série

```
>>> s2=s.copy()
>>>print(s2)
 0   1.3
 1   2.0
 2   5.0
 3   7.0
```

Création et consultation

Cas des Series

On peut également libeller les valeurs de `Series` :

```
>>> s=pandas.Series([1.3, 2, 5, 7], index = ['a', 'b', 'c', 'd'])
>>> print(s)
a    1.3
b    2.0
c    5.0
d    7.0
```

Création et consultation

Cas des Series

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
```

Création et consultation

Cas des Series

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
```

Création et consultation

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
```

Création et consultation

Cas des DataFrames

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
```

Création et consultation

Cas des DataFrames

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
```

Création et consultation

Cas des DataFrames

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
```

Création et consultation

Cas des DataFrames

```
>>> df1.loc['a2'] renvoie la Series de la ligne d'index 'a2'
A    2.7
B   10.0
C    5.4
D    7.0
```

Création et consultation

Cas des DataFrames

```
>>> 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
```

Création et consultation

Cas des DataFrames

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 les 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
```

Création et consultation

Cas des DataFrames

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]])
```

Création et consultation

Cas des DataFrames

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
```