Fichiers ?

Fichier - collection d'informations stockées sur une mémoire de masse (non volatile, capacité plus importante que la mémoire vive)

Types de fichiers – Ils se distinguent par…

1. Organisation des données : structuré, les informations sont organisées d'une manière particulière qu'il faut respecter (ex. fichiers d'enregistrements) ; non-structuré, les informations sont alignées à la suite (ex. fichier texte)
2. Mode d'accès aux données : accès indicé, utilisable comme un tableau ; accès séquentiel, lecture ou écriture pas à pas (ex. ligne par ligne dans un fichier texte)

La catégorisation n'est pas toujours très tranchée, un fichier XML par ex. est un fichier texte (manipulable avec un éditeur de texte) mais qui obéit à une organisation structurée

Accès séquentiel, non structuré

FICHIER TEXTE

Lecture en bloc avec read()

```
# -*- coding: utf -*-

#ouverture en lecture
f = open("voitures.txt","r")

#lecture
s = f.read()

#affichage
print("** contenu de s **")
print(s)
print("** fin contenu **")

#information sur s
print("type de s : ",type(s))
print("longueur de s : ", len(s))

#fermeture
f.close()
```

Fichier texte à lire :

« voiture.txt »

```
megane
clio
twingo
safrane
laguna
vel satis
```

- open() permet d'ouvrir un fichier, en lecture ici avec l'option « r ». La fonction renvoie un objet de type fichier stocké dans **f**
- le curseur de fichier est placé sur la première ligne
- read() lit tout le contenu du fichier d'un bloc
- close() ferme le fichier (et donc le déverrouille)

```
** contenu de s **
megane
clio
twingo
safrane
laguna
vel satis
** fin contenu **
type de s :  <class 'str'>
longueur de s :  43
```

s est de type « str », on se rend mieux compte en mode console:

```
>>> s
'megane\nclio\ntwingo\nsafrane\nlaguna\nvel satis'
```

\n représente le caractère spécial « saut de ligne » (line feed), c'est pour cela que print(s) fonctionne correctement.

Lecture en bloc avec readline**s** (avec « s »)

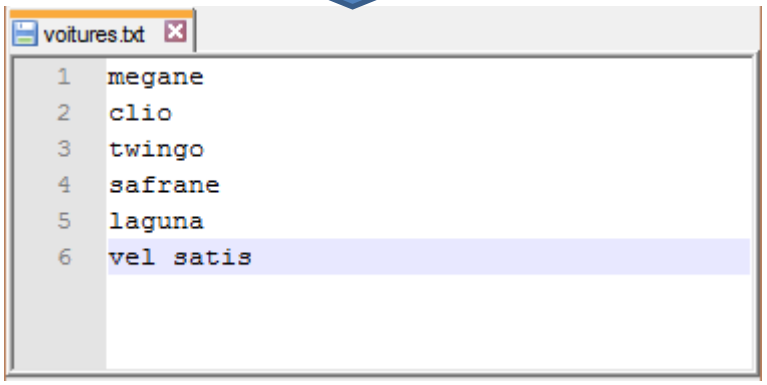
```
# -*- coding: utf -*-

#ouverture en lecture
f = open("voitures.txt","r")

#lecture
lst = f.readlines()

#affichage
print("** contenu de lst **")
print(lst)
print("** fin contenu **")

#information sur lst
print("type de s : ",type(lst))
print("longueur de s : ", len(lst))

#fermeture
f.close()
```

Le contenu du fichier est stocké dans une liste, une ligne = un élément. Le caractère \n est maintenu. Il n'est pas présent sur la dernière ligne de notre fichier exemple.

```
** contenu de lst **
['megane\n', 'clio\n', 'twingo\n', 'safrane\n', 'laguna\n', 'vel satis']
** fin contenu **
type de s :  <class 'list'>
longueur de s :  6
```

Remarque : saut de ligne ou pas sur la dernière ligne du fichier

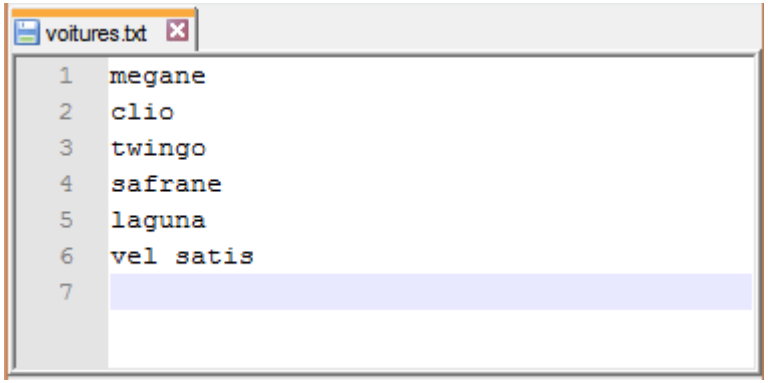
Ouvrir le fichier dans Notepad++



<u>Question :</u> Comment savoir s'il y a un saut de ligne ou pas à la dernière ligne de notre fichier texte ?

La ligne n°6 est la dernière ligne du fichier,
pas de saut ligne après « vel satis »

```
** contenu de lst **
['megane\n', 'clio\n', 'twingo\n', 'safrane\n', 'laguna\n', 'vel satis']
** fin contenu **
type de s :  <class 'list'>
longueur de s :  6
```



Il y a une ligne **vide** (la n°7) après « vel satis »

```
** contenu de lst **
['megane\n', 'clio\n', 'twingo\n', 'safrane\n', 'laguna\n', 'vel satis\n']
** fin contenu **
type de s :  <class 'list'>
longueur de s :  6
```

Lecture ligne par ligne avec readline (sans « s »)

```
# -*- coding: utf -*-

#ouverture en lecture
f = open("voitures.txt","r")

#lecture ligne itérativement
while True:
    s = f.readline()
    if (s != ""):
        print(s)
    else:
        break;

#fermeture
f.close()
```

- readline() lit la ligne courante et place le curseur sur la ligne suivante.
- la fonction renvoie la chaîne vide lorsque nous arrivons à la fin du fichier (Remarque : s'il y a une ligne blanche entre 2 véhicules, readline() renvoie le caractère « \n », ce n'est pas une chaîne vide).

```
megane

clio

twingo

safrane

laguna

vel satis
```

Il y a une ligne vide entre chaque véhicule parce que le caractère « \n » est toujours là, print() le prend en compte [Remarque : pour que print() n'en tienne pas compte, il faudrait faire print(s,end="")]

Lecture ligne par ligne en itérant sur l'objet fichier

```
# -*- coding: utf -*-

#ouverture en lecture
f = open("voitures.txt","r")

#lecture ligne itérativement
for s in f:
    print(s,len(s))

#fermeture
f.close()
```

C'est la forme la plus efficace – et la plus concise – pour une lecture ligne à ligne.

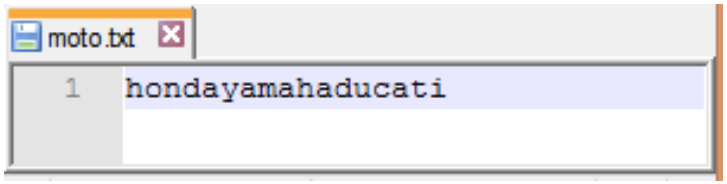
Le caractère \n est présent toujours, noter la longueur de la chaîne (+1 pour toutes sauf la dernière)

```
megane
 7
clio
 5
twingo
 7
safrane
 8
laguna
 7
vel satis 9
```

Ecriture d'un fichier texte avec write()

```
# -*- coding: utf -*-

#ouverture en écriture
f = open("moto.txt","w")

#écriture
f.write("honda")
f.write("yamaha")
f.write("ducati")

#fermeture
f.close()
```

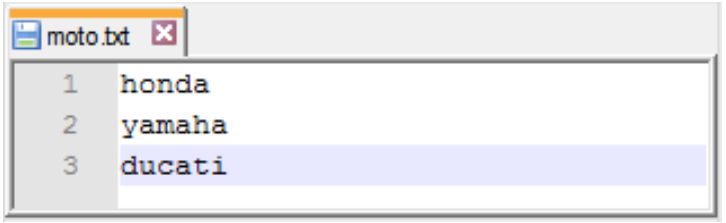
- open() permet d'ouvrir un fichier en écriture avec l'option « w ». La fonction renvoie un objet de type fichier référencée par **f**
- avec « w », le fichier est écrasé s'il existe déjà
- write() permet d'écrire la chaîne de caractères



Il manque les sauts de ligne pour distinguer chaque moto

```
# -*- coding: utf -*-

#ouverture en écriture
f = open("moto.txt","w")

#écriture
f.write("honda\n")
f.write("yamaha\n")
f.write("ducati")

#fermeture
f.close()
```

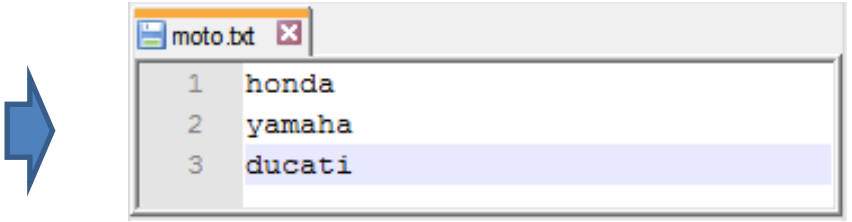
Nous insérons le caractère saut de ligne « \n » après chaque moto, sauf la dernière



Ecriture d'un fichier texte avec writelines()

```
# -*- coding: utf -*-

#ouverture en écriture
f = open("moto.txt","w")

#liste
lst = ["honda\n","yamaha\n","ducati"]

#écriture
f.writelines(lst)

#fermeture
f.close()
```

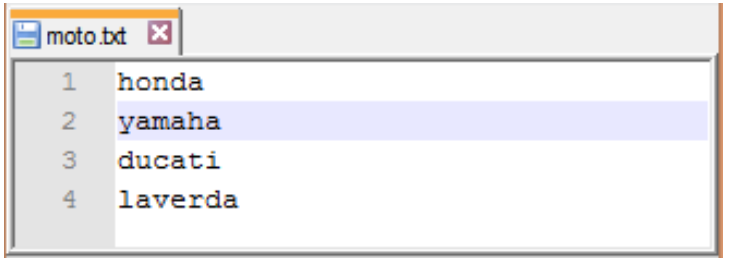
writelines() permet d'écrire directement le contenu d'une liste. Nous devons quand même insérer le caractère « \n » pour que le saut de ligne soit effectif dans le fichier.



Ajout dans un fichier texte

```
# -*- coding: utf -*-

#ouverture en ajout
f = open("moto.txt","a")

#ajouter un saut de ligne
f.write("\n")

#écriture
f.write("laverda")

#fermeture
f.close()
```

- open() avec l'option « a » permet d'ouvrir un fichier en mode ajout
- write() permet d'écrire la chaîne de caractères
- attention toujours au saut de ligne
- une ouverture en mode lecture / écriture est possible avec l'option « r+ » mais se positionner sur telle ou telle ligne pour y modifier ou insérer des informations est compliqué.



Traiter un fichier texte comme un fichier binaire (fichier d'octets)

FICHIER BINAIRE

Accès en mode binaire (1/2)

Python peut traiter un fichier en mode binaire, il peut lire octet par octet, ou par blocs d'octets. Un accès indicé est possible dans ce cas.
L'intérêt n'est pas flagrant pour un fichier texte, mais cette fonctionnalité ouvre la porte au traitement d'autres types de fichiers (ex. fichier image).

voitures.txt

```
megane
clio
twingo
safrane
laguna
vel satis
```

```
#ouverture en lecture
f = open("voitures.txt","rb")
#lire un octet
a = f.read(1)
print(a)
#type de a → array de bytes
print(type(a))
#transformer en chaîne de caractères
s = a.decode("utf-8")
print(s)
print(type(s))
#lire une 2nde fois
a = f.read(1)
print(a)
#pos. du curseur
print("position : ",f.tell())
```

- option « **rb** » pour open() pour lecture **et** mode binaire
- read() prend un paramètre : nb. d'octets à lire
- decode() permet de transformer le tableau d'octets en chaîne de caractères, « utf-8 » est le mode d'encodage
- après un read(), le curseur de fichier se place sur l'octet suivant, on peut connaître la position du curseur avec tell() [début du fichier = indice 0]

```
b'm'
<class 'bytes'>
m
<class 'str'>
b'e'
position :  2
```

b pour indiquer qu'il s'agit d'un tableau de bytes

Accès en mode binaire (2/2)

```
#positionner le curseur
f.seek(0,0)
#lire un bloc d'octets
a = f.read(6)
print(a)
print("longueur = ",len(a))
#aller à l'octet n°5
#à partir du début
f.seek(5,0)
a = f.read(1)
print(a)
#lire le dernier octet
f.seek(-1,2)
a = f.read(1)
print(a)
#fermeture
f.close()
```

voitures.txt

```
megane
clio
twingo
safrane
laguna
vel satis
```

- seek() permet de positionner le curseur, le 1er paramètre est la position, le 2nd est la référence : 0 à partir du début du fichier, 2 à partir de la fin, 1 à partir de la position courante
- noter le seek(-1,2) avec un indice négatif, comme pour les listes ou les tuples

```
b'megane'
longueur =  6
b'e'
b's'
```

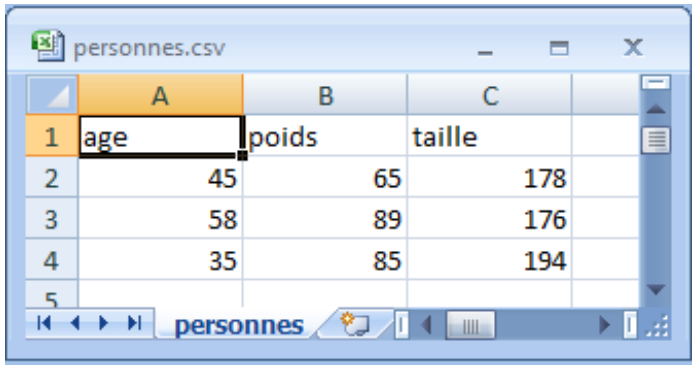
Fichier structuré

FORMAT CSV

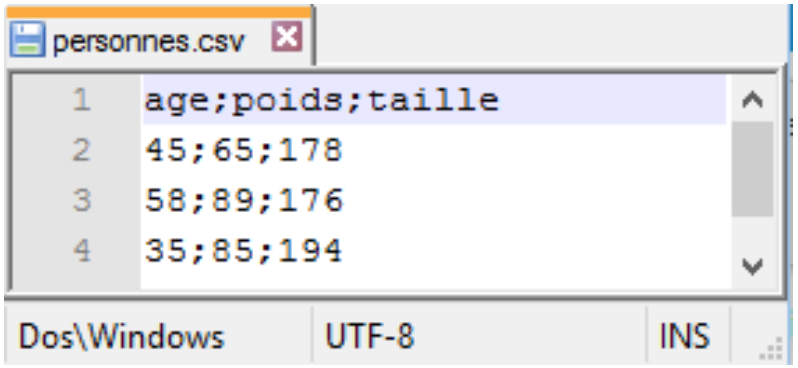
Format CSV

Un fichier CSV (Comma-separated values) est un fichier texte (!) avec une structure **tabulaire**. Ce type de fichier peut être généré à partir d'un tableur (Excel ou Calc d'Open/Libre Office). C'est un format privilégié pour le traitement statistique des données.

Excel



Notepad++



Remarques : (1) Le passage d'une ligne à l'autre est matérialisé par un saut de ligne ; (2) ";" est utilisé comme séparateur de colonnes (paramétrable, ça peut être tabulation "\t" aussi souvent) ; (3) le point décimal dépend de la langue (problème potentiel pour les conversions) ; (4) la première ligne joue le rôle d'en-tête de colonnes souvent (nom des variables en statistique).

Lecture du format CSV – Structure de liste

```
#ouverture en lecture
f = open("personnes.csv","r")

#importation du module csv
import csv

#lecture - utilisation du parseur csv
lecteur = csv.reader(f,delimiter=";")

#affichage - itération sur chaque ligne
for ligne in lecteur:
    print(ligne)

#fermeture du fichier
f.close()
```

Paramétrage du séparateur de colonnes avec l'option delimiter.

Chaque ligne est une liste.

```
['age', 'poids', 'taille']
['45', '65', '178']
['58', '89', '176']
['35', '85', '194']
```

Remarques :

1. La première ligne est une observation comme une autre.
2. Toutes les valeurs sont considérées comme chaîne de caractères [une conversion automatique des chiffres est possible, mais elle ne fonctionne pas si le point décimal est « , » - mieux vaut une conversion explicite avec float()]

Lecture du format CSV – Structure de dictionnaire

```
#ouverture en lecture
f = open("personnes.csv","r")

#importation du module csv
import csv

#lecture
lecteur = csv.DictReader(f,delimiter=";")

#affichage
for ligne in lecteur:
    print(ligne)

#fermeture
f.close()
```

Chaque ligne est un **Dict**

```
{'poids': '65', 'age': '45', 'taille': '178'}
{'poids': '89', 'age': '58', 'taille': '176'}
{'poids': '85', 'age': '35', 'taille': '194'}
```

Remarques :

1. La première ligne est reconnue comme nom de champs
2. On utilise les clés pour accéder aux valeurs. Pour un accès indicé, une solution possible serait de convertir la collection des valeurs [ligne.values()] en liste.

Fichier structuré

FORMAT JSON

Format JSON

Un fichier JSON (JavaScript Object Notation) est un fichier texte (!) mais avec une structure standardisée permettant de rendre compte de l'organisation des données. C'est un format reconnu pour l'échange de données entre applications. Deux types d'éléments structurels : (1) des ensembles de paires « nom – valeur » ; (2) des listes ordonnées de valeur.

```
#début définition
class Personne:
    """Classe Personne"""
    #constructeur
    def __init__(self):
        #lister les champs
        self.nom = ""
        self.age = 0
        self.salaire = 0.0
    #fin constructeur
    … saisie et affichage …
#fin définition
```

A faire :

(A) Comment sauvegarder un objet de type Personne dans un fichier ?

(B) Comment sauvegarder une collection (liste) de personnes.

Exemples

personne.txt

```
1  {"Age": 34, "Salaire": 1500.0, "Nom": "Toto"}
```

personnes.txt

```
1  [{"Age": 34, "Nom": "Toto", "Salaire": 1250.0},
   {"Age": 43, "Nom": "Tata", "Salaire": 2600.0}]
```

Le plus simple : passer par le type dictionnaire

Idée : Le type dict permet de définir une collection d'éléments prenant la forme « étiquette : valeur ». Nous exploitons cela pour sauver les champs de l'objet.

```
# -*- coding: utf -*-
#import Personne
import ModulePersonne as MP
#import module json
import json
#saisie personne
p = MP.Personne()
p.saisie()
#sauvegarde
f = open("personne.json","w")
#dictionnaire
d = {"Nom":p.nom,"Age":p.age,"Salaire":p.salaire}
#sauver au format json
json.dump(d,f)
#fermer le fichier
f.close();
```

- noter l'importation du module json
- une étape clé est la transformation de l'objet référencé par **p** en un dictionnaire référencé par **d**
- la fonction dump() de la classe json permet de stocker l'objet dictionnaire dans le fichier référencé par **f**
- il ne faut pas oublier d'ouvrir en écriture puis de fermer le fichier
- noter le format de fichier json avec les accolades {} pour délimiter un enregistrement

personne.txt

```
1  {"Age": 34, "Salaire": 1500.0, "Nom": "Toto"}
```

Remarque : Passer par un dictionnaire est un artifice destiné à nous faciliter la vie. De manière plus académique, il faudrait rendre l'objet directement sérialisable c.-à-d. quand on fait dump() dessus, les informations sont correctement inscrites dans le fichier. A voir en M2 mon cours de C#.

Sauvegarde : astuce, utiliser l'attribut standard __dict__

```
# -*- coding: utf -*-
#import Personne
import ModulePersonne as MP
#import module json
import json
#saisie personne
p = MP.Personne()
p.saisie()
#sauvegarde
f = open("personne.json","w")
#sauver au format json
json.dump(p.__dict__,f) #!!!
#fermer le fichier
f.close();
```

Idée : Tous les objets Python (instance de classe) possède l'attribut standard __dict__, il recense les champs de la classe et leurs valeurs pour l'objet. Il est de type dictionnaire. C'est exactement ce qu'il nous faut.

← le code est grandement simplifié

Charger l'objet via le type dictionnaire

```
# -*- coding: utf -*-
#import Personne
import ModulePersonne as MP
#import module json
import json
#ouverture fichier
f = open("personne.json","r")
#chargement
d = json.load(f)
print(d)
print(type(d))
#transf. en Personne
p = MP.Personne()
p.nom = d["Nom"]
p.age = d["Age"]
p.salaire= d["Salaire"]
#affichage
p.affichage()
#fermeture
f.close();
```

- le fichier est ouvert en lecture maintenant
- load() de la classe json s'occupe du chargement
- l'objet chargé est de type dict (dictionnaire)
- nous créons une instance de Personne, et nous recopions les informations
- l'objet référencé par p peut vivre sa vie par la suite Ex. ici utilisation de la méthode affichage()

```
{'Salaire': 1500.0, 'Nom': 'Toto', 'Age': 34}
<class 'dict'>
Son nom est  Toto
Son âge :  34
Son salaire :  1500.0
```

Sauver une liste d'objets

Objectif : Sauvegarder un ensemble de personnes dans un fichier JSON.

Idée : Utiliser une double collection. Une liste (list) dans laquelle sont insérés des dictionnaires (dict).

```
#import Personne
import ModulePersonne as MP
#import module json
import json
#liste vide
liste = []
#nb. de pers ?
n = int(input("Nb de pers : "))
#saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)
#sauvegarde
f = open("personnes.json","w")
#créer une liste temporaire
tmp = []
#pour chaque personne
for p in liste:
    #créer un dictionnaire
    d = {}
    d["Nom"] = p.nom
    d["Age"] = p.age
    d["Salaire"] = p.salaire
    #ajouter dans liste tmp
    tmp.append(d)
#sauvegarde de la liste tmp
json.dump(tmp,f)
#fermer le fichier
f.close();
```

- format JSON : noter l'imbrication des [ ] et { } pour délimiter la liste et chaque enregistrement
- effectuer un dump() sur la liste principale tmp revient à sauver chaque dictionnaire (Personne) qui le compose.

```
Nb de pers : 2
Nom : Toto
Age : 34
Salaire : 1250
Nom : Tata
Age : 43
Salaire : 2600
```

← Un exemple

personnes.txt

```
1  [{"Age": 34, "Nom": "Toto", "Salaire": 1250.0},
   {"Age": 43, "Nom": "Tata", "Salaire": 2600.0}]
```

Charger une liste d'objets

```
#import Personne
import ModulePersonne as MP

#import module json
import json

#ouverture fichier
f = open("personnes.json","r")

#chargement
tmp = json.load(f)

#conv. en liste de personnes
liste = []
for d in tmp:
    #créer une personne
    p = MP.Personne()
    p.nom = d["Nom"]
    p.age = d["Age"]
    p.salaire= d["Salaire"]
    #affichage
    p.affichage()
    #l'ajouter dans la liste
    liste.append(p)

print("Nb personnes : ",len(liste))

#fermeture
f.close();
```

Objectif : Charger un ensemble de personnes à partir d'un fichier JSON.

Idée : Convertir les dict en objet de type Personne

Effectuer un load() permet de charger la liste de dictionnaires, référencée par tmp. Une instance de Personne est créée pour chaque élément de type dict, les informations sont recopiées.

```
Son nom est  Toto
Son âge :  34
Son salaire :  1250.0
Son nom est  Tata
Son âge :  43
Son salaire :  2600.0
Nb personnes :  2
```

Fichier structuré

FORMAT XML

Format XML

Un fichier XML (Extensible Markup Language) est un fichier texte (!) permettant de décrire des documents. Le principe de construction est simple et immuable (organisation hiérarchique, balises, attributs), mais la structure est évolutive (extensible) en fonction du document à décrire. C'est aussi un format reconnu pour l'échange de données entre applications.

Structure pour un objet

```
<?xml version="1.0"?>
- <Personne>
    <Nom>Toto</Nom>
    <Age>35</Age>
    <Salaire>2100.0</Salaire>
  </Personne>
```

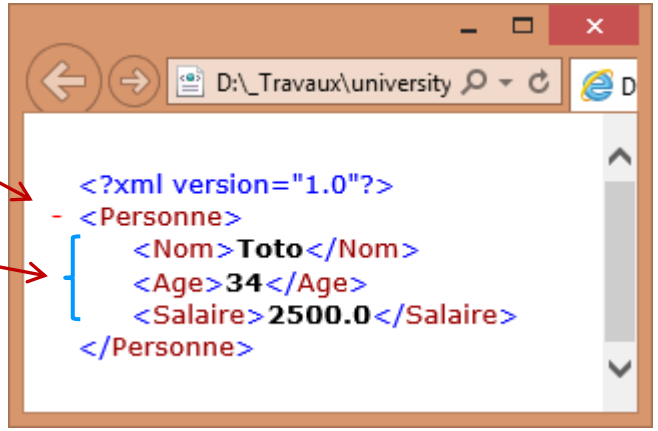
Structure pour une liste d'objets

```
<?xml version="1.0"?>
- <Individus>
  - <Personne>
      <Nom>Toto</Nom>
      <Age>35</Age>
      <Salaire>1200.0</Salaire>
    </Personne>
  - <Personne>
      <Nom>Tata</Nom>
      <Age>36</Age>
      <Salaire>3200.0</Salaire>
    </Personne>
  </Individus>
```

Créer un fichier XML avec un objet

```
#import Personne
import ModulePersonne as MP

#import module xml
import xml.etree.ElementTree as ET

#saisie personne
p = MP.Personne()
p.saisie()

#écriture racine
root = ET.Element("Personne")

#écriture des élements
item_nom = ET.SubElement(root,"Nom")
item_nom.text = p.nom

item_age = ET.SubElement(root,"Age")
item_age.text = str(p.age)

item_sal = ET.SubElement(root,"Salaire")
item_sal.text = str(p.salaire)

#sauvegarde fichier
tree = ET.ElementTree(root)
tree.write("personne.xml")
```

- le module « xml » permet de gérer les fichiers au format XML sous Python
- noter la nature hiérarchique de la structure
- root est le **nœud racine** de la structure
- il peut y avoir des nœuds intermédiaires. cf. la sauvegarde des listes d'objets
- nous typons toutes les informations en chaîne de caractères cf. **str()**
- **Personne, Nom, Age, Salaire** sont des balises XML

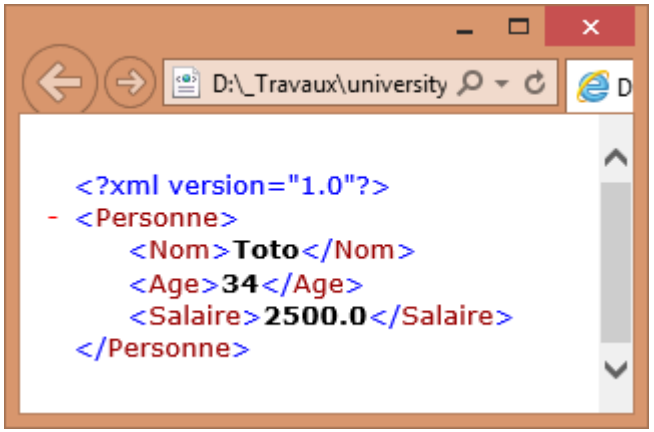


← Il n'est pas nécessaire de passer par l'objet fichier

Charger un fichier XML

```
#import Personne
import ModulePersonne as MP

#import module xml
import xml.etree.ElementTree as ET

#ouverture
tree = ET.parse("personne.xml")

#récupérer la racine
root = tree.getroot()

#trans. en Personne
p = MP.Personne()
p.nom = root.findtext("Nom")
p.age = int(root.findtext("Age"))
p.salaire= float(root.findtext("Salaire"))

#affichage
p.affichage()
```

- parse() ouvre le fichier et charge la structure (l'arbre)
- getroot() donne accès à la racine de l'arbre
- findtext() permet de lire le contenu des balises ici [sous le nœud racine, **root**.findtext()]

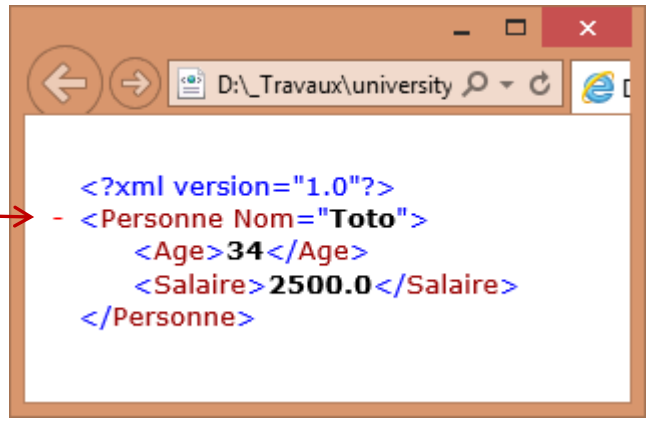



```
Son nom est  Toto
Son âge :  34
Son salaire :  2500.0
```

Créer un fichier XML – Variante avec les attributs

```
#import Personne
import ModulePersonne as MP

#import module xml
import xml.etree.ElementTree as ET

#saisie personne
p = MP.Personne()
p.saisie()

#écriture racine
root = ET.Element("Personne")

#associer un attribut à la balise Personne
root.set("Nom",p.nom)

item_age = ET.SubElement(root,"Age")
item_age.text = str(p.age)

item_sal = ET.SubElement(root,"Salaire")
item_sal.text = str(p.salaire)

#sauvegarde fichier
tree = ET.ElementTree(root)
tree.write("personne.xml")
```

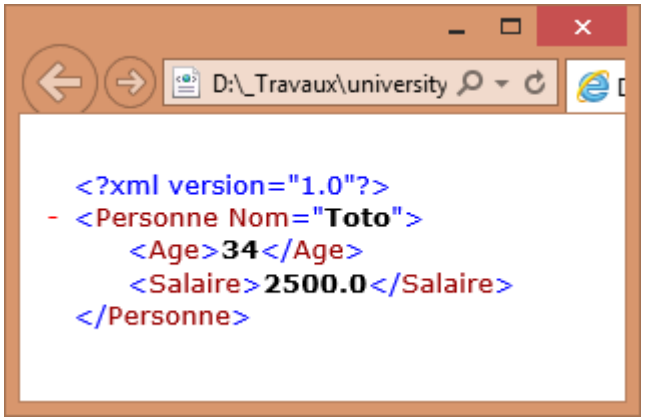
- on peut associer des attributs aux balises
- voici une structure possible pour le fichier XML
- l'attribut « Nom » est associé à la balise « Personne »



Charger un fichier XML avec attributs

```
#import Personne
import ModulePersonne as MP

#import module xml
import xml.etree.ElementTree as ET

#ouverture
tree = ET.parse("personne.xml")

#récupérer la racine
root = tree.getroot()

#trans. en Personne
p = MP.Personne()
p.nom = root.get("Nom")
p.age = int(root.findtext("Age"))
p.salaire= float(root.findtext("Salaire"))

#affichage
p.affichage()
```

Accès à l'attribut associé à la racine




```
Son nom est  Toto
Son âge :  34
Son salaire :  2500.0
```

Liste des commandes disponibles sur un nœud de la structure

Les méthodes des noeuds

Si vous voulez approfondir vos connaissances sur la lib lxml, il existe plein d'autres méthodes associées aux noeuds que vous pouvez voir en lançant la commande help(*<noeuds>*). L'aide est en anglais et comme je suis sympa, je vous l'ai traduite:

```
addnext(element)      : Ajoute un élément en tant que frère juste après l'élément
addprevious(element)  : Ajoute un élément en tant que frère juste avant l'élément
append(element)       : Ajoute un sous-élément à la fin de l'élément
clear()               : Supprime tous les sous-éléments
extends(elements)     : Etend les éléments passé en paramètre
find(path)            : Recherche le premier sous-élément qui correspond au tag/path
findall(path)         : Recherche tous les sous-éléments qui correspondent au tag/path
findtext(path)        : Trouve le texte du premier sous-élément qui correspond au tag/path
get(key)              : Recupère l'attribut d'un élément
getchildren()         : Retourne tous les enfants d'un élément ! attention déprécié pour list(elemen
getnext()             : Retourne le prochain élément frère
getparent()           : Retourne le parent de l'élément
getprevious()         : Retourne l'élément frère précédant
index(child)          : Trouve la position de l'élément
insert(index)         : Insère un sous-élément à la position indiquée
items()               : Retourne les attributs d'un élément (dans un ordre aléatoire)
keys()                : Retourne une liste des noms des attributs
remove(element)       : Supprime l'élément passé en paramètre
replace(el1, el2)     : Remplace el1 par el2
set(key, value)       : Créer un attribut avec une valeur
values()              : Retourne les valeurs des attributs
xpath(path)           : Evalue une expression xpath
```

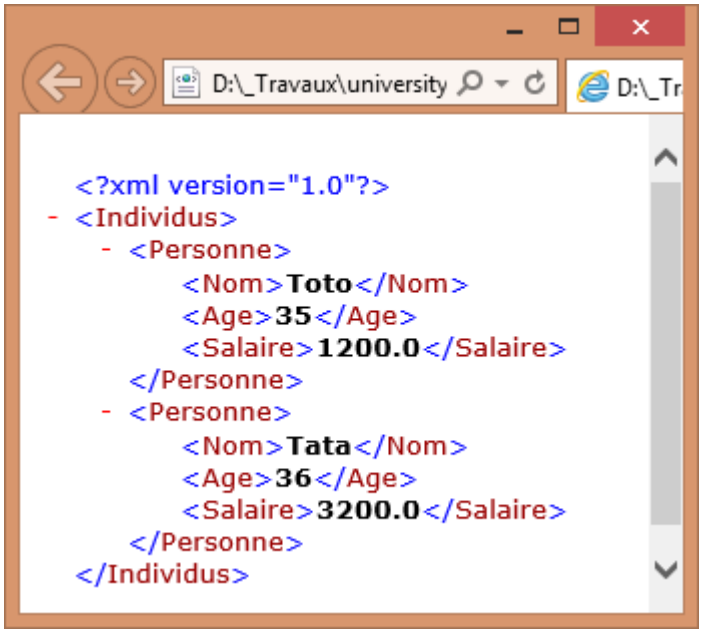
Source : http://apprendre-python.com/

Voir aussi : http://lxml.de/tutorial.html

Sauvegarde XML d'une liste d'objets

```
#import Personne
import ModulePersonne as MP
#import module xml
import xml.etree.ElementTree as ET
#liste vide
liste = []
#nb. de pers ?
n = int(input("Nb de pers : "))
#saisie liste
for i in range(0,n):
    a = MP.Personne()
    a.saisie()
    liste.append(a)
#racine
root = ET.Element("Individus")
#pour chaque personne
for p in liste:
    #personne xml
    px = ET.SubElement(root,"Personne")
    #champs
    item_nom = ET.SubElement(px,"Nom")
    item_nom.text = p.nom
    item_age = ET.SubElement(px,"Age")
    item_age.text = str(p.age)
    item_sal = ET.SubElement(px,"Salaire")
    item_sal.text = str(p.salaire)
#sauvegarde
tree = ET.ElementTree(root)
tree.write("personnes.xml")
```

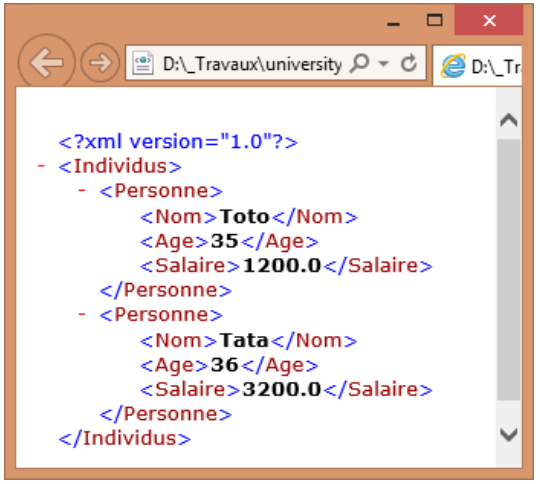
- il y a deux niveaux dans la structure
- px (Personne) est un sous-élément de root (Individus)
- c'est bien sur px que l'on insère les informations sur chaque personne



Chargement d'une liste d'objets à partir d'un fichier XML

```
#import Personne
import ModulePersonne as MP
#import module xml
import xml.etree.ElementTree as ET

#ouverture
tree = ET.parse("personnes.xml")

#récupérer la racine
root = tree.getroot()

#conv. en liste de personnes
liste = []
for child in root:
    #créer une personne
    p = MP.Personne()
    p.nom = child.findtext("Nom")
    p.age = int(child.findtext("Age"))
    p.salaire= float(child.findtext("Salaire"))
    #affichage
    p.affichage()
    #l'ajouter dans la liste
    liste.append(p)

print("Nb personnes : ",len(liste))
```

- On explore les deux niveaux lors du chargement
- root <Individus> se comporte comme une liste (cf. la boucle for)
- chaque élément child correspond au nœud <Personne>




```
Son nom est  Toto
Son âge :  35
Son salaire :  1200.0
Son nom est  Tata
Son âge :  36
Son salaire :  3200.0
Nb personnes :  2
```