



UNIVERSITÉ  
CÔTE D'AZUR

# Bases de l'informatique 1

## Cours 8. Ensembles, dictionnaires et fichiers texte

---

Olivier Baldellon

Courriel : `prenom.nom@univ-cotedazur.fr`

Page professionnelle : <https://upinfo.univ-cotedazur.fr/~obaldellon/>

LICENCE I — FACULTÉ DES SCIENCES ET INGÉNIERIE DE NICE — UNIVERSITÉ CÔTE D'AZUR

- 🍃 Partie I. Ensembles
- 🍃 Partie II. Fonctions de hachage
- 🍃 Partie III. Dictionnaires
- 🍃 Partie IV. Mémoïsation
- 🍃 Partie V. Compléments sur les chaînes
- 🍃 Partie VI. Systèmes de fichier
- 🍃 Partie VII. E/S : écrire dans un fichier
- 🍃 Partie VIII. E/S : lire dans un fichier

- ▶ Un ensemble en Python est une collection finie d'objets
  - ▶ Une collection **sans répétition** et **sans ordre**
- ▶ Un ensemble **n'est pas une séquence!**
  - ▶ On ne peut pas accéder aux éléments via des indices. `E[i]`
- ▶ Ils sont notés avec des accolades comme en mathématiques.

```
>>> { 1, 2, 3, 1, 2 } # Ni répétition
{1, 2, 3}
>>> { 1, 2, 3 } == { 3, 1, 2 } # ni ordre
True
>>> {False, 'bleu', 2, 'bleu'} == {2, False, 'bleu', 2, 2}
True
```

SHELL

- ▶ L'ensemble vide est noté : `set()` (et non pas `{}` qui est un dictionnaire)

```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
>>> set()
set()
```

SHELL

- ▶ La notation  $E[i]$  n'a pas de sens
  - ▶ Les éléments ne sont pas ordonnés : ils n'ont donc pas d'indice.

```
>>> E = { 22, 31.2, 'Salut', True }
>>> E[2]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

SHELL

- ▶ On peut parcourir un ensemble avec une boucle for :
  - ▶ L'ordre n'est pas respecté (car il n'y a pas d'ordre!)

```
>>> for x in E:
...     print(x)
True
Salut
22
31.2
```

SHELL

- ▶ L'opérateur `in` permet de savoir si un élément **appartient** à un ensemble

```
>>> A = {22, 'Salut'} ; B = { 22, 31.2, 'Salut', True }
>>> print( 31.2 in A , 31.2 in B)
False True
```

- ▶ En mathématiques,  $A$  est **inclus** dans  $B$  si tout élément de  $A$  appartient à  $B$ 
  - ▶  $A \subseteq B \equiv \forall x \in A, x \in B$

- ▶ Traduisons cela en Python avec une boucle `for`

```
def inclusion(A,B):
    for x in A:
        if not(x in B):
            return False
    return True
```

```
>>> inclusion(A,B)
True
>>> inclusion(B,A)
False
>>> inclusion(A,A)
True
```

- ▶ Ou directement avec l'opérateur `<=` (si on veut l'inclusion stricte : `<`) :

```
>>> print( A<=B , B<=A , A<=A , A<A , set()<A)
True False True False True
```

- ▶ En mathématiques, le **cardinal** est le nombre d'éléments d'un ensemble.

```
def cardinal(E):
    c = 0
    for x in E:
        c=c+1
    return c
```

SCRIPT

```
>>> cardinal({2,2,1,1,3,3,2,1,2,3})
```

```
3
```

```
>>> {2,2,1,1,3,3,2,1,2,3} # ne contient bien que 3 éléments
{1, 2, 3}
```

```
>>> cardinal(set())
```

```
0
```

SHELL

- ▶ Comme d'habitude, on ~~s'embête~~ pour rien : fonction **len**
  - ▶ C'est **important pédagogiquement** de savoir réécrire les fonctions de base.

```
>>> len({2,2,1,1,3,3,2,1,2,3})
```

```
3
```

```
>>> len(set())
```

```
0
```

SHELL

- ▶ On peut construire un ensemble en donnant directement ses valeurs

```
>>> E = {1 , 2 , 3 , 4}
>>> E
{1, 2, 3, 4}
```

SHELL

- ▶ De même que les listes, on peut les construire par compréhension.

```
>>> L = [ x%10 for x in range(100) if x%6==0 ]
>>> L # L est une liste
[0, 6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96]
>>> E = { x%10 for x in range(100) if x%6==0 }
>>> E # E est un ensemble
{0, 2, 4, 6, 8}
```

SHELL

- ▶ On peut utiliser la fonction de conversion `set` (voir cours 5 page 18)

```
>>> set('abc')
{'c', 'b', 'a'}
>>> set(['a', 'b', 'c'])
{'c', 'b', 'a'}
>>> set(('a', 'b', 'c'))
{'c', 'b', 'a'}
```

SHELL

- ▶ Les ensembles sont mutables : Cela signifie que l'on peut les modifier.
- ▶ On peut ajouter un élément avec la méthode `add`

```

>>> E          # E contient 3 éléments
{1, 2, 3}
>>> E.add(12)
>>> E          # E contient maintenant 4 éléments
{1, 2, 3, 12}
>>> E.add(1)
>>> E          # E contient toujours 4 éléments
{1, 2, 3, 12}
    
```

SHELL

- ▶ On peut supprimer un élément avec la méthode `remove`

```

>>> E = {'Salut', False, (1,2,3), 'Ha ha ha'}
>>> E.remove('HA ha ha')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError: 'HA ha ha'
>>> E
{False, 'Ha ha ha', 'Salut', (1, 2, 3)}
>>> E.remove('Ha ha ha')
    
```

SHELL

- ▶ On veut écrire l'union de deux ensembles.

```
def union(A,B):
    E = set() # ensemble vide
    for x in A:
        E.add(x)
    for x in B:
        E.add(x)
    return E
```

SCRIPT

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> union(A,B)
{1, 2, 3, 4, 6, 8}
>>> union(A,A)
{1, 2, 3, 4}
```

SHELL

- ▶ Comme toujours, la fonction existe déjà sous Python!
  - ▶ L'**union**  $A \cup B = \{x : x \in A \text{ ou } x \in B\}$  se note  $A \mid B$
  - ▶ L'**intersection**  $A \cap B = \{x : x \in A \text{ et } x \in B\}$  se note  $A \& B$
  - ▶ La **différence**  $A \setminus B = \{x : x \in A \text{ et } x \notin B\}$  se note  $A - B$

```
>>> A={1,2,3,4}
>>> B={2,4,6,8}
>>> A|B
{1, 2, 3, 4, 6, 8}
```

SHELL

```
>>> A&B
{2, 4}
>>> A-B
{1, 3}
```

SHELL

- ▶ Exercice : écrire l'intersection et la différence sans utiliser d'opérateurs.

- ▶ Bonne question : essayons!

```
>>> { '123' , 'abc' } # str : immuable
{'123', 'abc'}
>>> { (1,2,3) , ('a','b','c') } # tuple : immuable
{(1, 2, 3), ('a', 'b', 'c')}
>>> { [1,2,3] , ['a','b','c'] } # list : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> { {1,2,3} , {'a','b','c'} } # set : mutable
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unhashable type: 'set'
```

SHELL

- ▶ Un ensemble est **mutable**, mais ses éléments doivent être **immuable**.
  - ▶ On peut faire des ensembles de tuple (ensembles de points du plan)
  - ▶ On ne peut pas faire des ensembles de listes ou d'ensemble.
- ▶ Que signifie *unhashable type* du message d'erreur?
  - ▶ En interne, les ensembles utilisent des fonctions de hachage.

- 🍃 Partie I. Ensembles
- 🍃 Partie II. Fonctions de hachage
- 🍃 Partie III. Dictionnaires
- 🍃 Partie IV. Mémoïsation
- 🍃 Partie V. Compléments sur les chaînes
- 🍃 Partie VI. Systèmes de fichier
- 🍃 Partie VII. E/S : écrire dans un fichier
- 🍃 Partie VIII. E/S : lire dans un fichier

- ▶ Qu'est-ce qu'une empreinte digitale ?
  - ▶ C'est un **marqueur** biologique en théorie **unique**.
  - ▶ Facile à stocker, « facile » à comparer, facile à obtenir
- ▶ Existe-il un équivalent numérique ?
  - ▶ Une **fonction de hashage** permet de construire une empreinte numérique.
- ▶ Exemple : la fonction md5
  - ▶ Elle calcule une empreinte (*digest*) de 128 bits.
  - ▶ On représente l'empreinte le plus souvent par son écriture hexadécimale



```
>>> from hashlib import md5
>>> def h(x):
...     return md5(repr(x).encode()).hexdigest()
>>> h(3)
'eccbc87e4b5ce2fe28308fd9f2a7baf3'
>>> h([1,2,3])
'49a5a960c5714c2e29dd1a7e7b950741'
>>> h(min)
'0f61485fd84d673c233e28e1d2a5acfe'
```

SHELL

- ▶ À quoi ces fonctions servent-elles ?
  - ▶ Elles permettent de créer un **identifiant** pour un objet quelconque.
  - ▶ En particulier cela sert à **comparer des données** volumineuses.
  - ▶ Sauriez-vous trouver la différence entre les deux chaînes ?
  - ▶ Grâce à la fonction de hachage, la non-égalité est immédiate.

```
>>> A='123456789101112131415161718192021222324252627'  
>>> B='123456789101112131415161718192021222324252627'  
>>> h(A)  
'c42012567482404030e362f0c3813c15'  
>>> h(B)  
'e5a41e2edb893242fe25aad75dfcca66'
```

SHELL

- ▶ Soit  $s_1$  et  $s_2$  deux chaînes et  $h(s_1)$  et  $h(s_2)$  leurs empreintes.
- ▶ On a la **garantie** suivante :
  - ▶ Si  $h(s_1) \neq h(s_2)$  alors forcément  $s_1 \neq s_2$  (car  $h$  est une fonction)
- ▶ Les propriétés suivantes sont **extrêmement probables** :
  - ▶ Si  $h(s_1) = h(s_2)$  on peut en pratique considérer que  $s_1 = s_2$  (on a 1 chance sur 340 282 366 920 938 463 463 374 607 431 768 211 455 de se tromper)
  - ▶ Si  $s_1$  et  $s_2$  sont très proches,  $h(s_1)$  et  $h(s_2)$  sont très différents.

- ▶ Supposons que l'on souhaite implémenter les ensembles par des listes
  - ▶ On réimplémente les ensembles pour des questions pédagogiques

```
def ajouter(L,x):  
    for e in L:  
        if x == e: return # terminaison car x est déjà dans L  
    L.append(x) # sinon, on ajoute x  
    return
```

SCRIPT

- ▶ Pour ajouter un élément, je dois comparer avec tous les éléments.
  - ▶ S'il y a  $n$  éléments de taille  $T$ , il faudra faire  $n \cdot T$  comparaisons.
- ▶ Pour gagner en efficacité, on stocke chaque élément avec son empreinte.

```
def ajouter(L,x):  
    hx=h(x) # je calcule l'empreinte de x  
    for e in L:  
        (hy,y) = e # y est stocké avec son empreinte  
        if hx == hy: return # le programme termine  
    L.append((hx,x))  
    return
```

SCRIPT

- ▶ Dorénavant je dois faire seulement  $N$  comparaisons d'empreintes.

- ▶ Dans Python
  - ▶ Les ensembles sont implémentés de manière bien plus élaborées.
  - ▶ Mais ils utilisent des tables de hachage.
  - ▶ L'ajout d'élément ne dépend pas de la taille des éléments de E.
  
- ▶ En informatique en général : on utilise les empreintes
  - pour vérifier qu'un téléchargement correspond au bon fichier
    - ▶ On compare les empreintes (*MD5 check sum*)
    - ▶ Si l'empreinte est bonne, on a bien une version correcte du bon fichier
  
  - pour stocker des mots de passe sans les révéler.
    - ▶ On stocke les empreintes
    - ▶ On compare avec l'empreinte du mot de passe fourni par l'utilisateur.
    - ▶ À aucun moment on ne stocke les mots de passe
    - ▶ Une empreinte ne permet pas de retrouver le mot de passe
  
  - pour certifier la liste chaînée d'une *blockchain* (bitcoin) ou de git.

- Partie I. Ensembles
- Partie II. Fonctions de hachage
- Partie III. Dictionnaires
- Partie IV. Mémoïsation
- Partie V. Compléments sur les chaînes
- Partie VI. Systèmes de fichier
- Partie VII. E/S : écrire dans un fichier
- Partie VIII. E/S : lire dans un fichier

- ▶ Un dictionnaire est une collection de couples clé:valeur.
  - ▶ clé est forcément non mutable;
  - ▶ valeur peut être modifiée.

```
>>> mon_panier = {'pommes':243 , 'poires':123 }  
>>> mon_panier  
{'pommes': 243, 'poires': 123}
```

SHELL

- ▶ On accède aux valeurs en utilisant les clés comme indices.

```
>>> mon_panier['poires']  
123  
>>> mon_panier['pommes']  
243
```

SHELL

- ▶ Un dictionnaire vide se note {} ou mieux dict()
- ▶ Toutes les clés doivent être distinctes

```
>>> {'pommes':243 , 'poires':123, 'pommes':23 }  
{'pommes': 23, 'poires': 123}
```

SHELL

- ▶ L'accès à une valeur est extrêmement rapide.
  - ▶ C'est le principal intérêt des dictionnaires.
  - ▶ Les dictionnaires utilisent des tables de hachage
- ▶ La recherche est **unidirectionnelle**
  - ▶ on va de la clé vers les valeurs
- ▶ La clé doit être non mutable.
  - ▶ Typiquement des chaînes et des nombres.
  - ▶ On peut utiliser des tuples ne contenant que des éléments non mutables.

```
>>> dico = { 'un':234 , 2:[3,4,5] , 3.5:'Bonjour' }
>>> dico[3.5]
'Bonjour'
>>> dico['un']
234
>>> dico['Bonjour'] # 'Bonjour' est une valeur !
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError: 'Bonjour'
```

SHELL

- ▶ Si on utilise une clé qui n'existe pas, une exception est levée.
  - ▶ Pour savoir si une clé existe on peut utiliser mot-clé `in`

```
>>> partiel = { 'Alice':15 , 'Bob':13 , "Charline":9}
>>> partiel["Gustave"]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError: 'Gustave'
>>> 'Gustave' in partiel # "Gustave" est-il une clé ?
False
```

SHELL

- ▶ On veut affecter à note la valeur associée à l'étudiant étu
  - ▶ Si une telle clé n'existe pas, on pose `note="ABS"`

```
try:
    note = partiel[étu]
except KeyError:
    note = "ABS"
```

SCRIPT

```
if étu in partiel:
    note = partiel[étu]
else:
    note = "ABS"
```

SCRIPT

- ▶ Ou plus simplement en une ligne avec la méthode `get`

```
>>> partiel.get("Alice", 'ABS')
15
>>> partiel.get("Gustave", 'ABS')
'ABS'
```

SHELL

- ▶ Un dictionnaire est **mutable** : il est **modifiable**.
- ▶ On peut modifier la valeur associée à une clé en utilisant l'**affectation**.

```
>>> dico = { 11: 'unu' , 22:'Du' , 33:'tri' }
>>> dico[22]='du'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
```

SHELL

- ▶ On peut ajouter un nouveau couple clé:valeur en utilisant l'**affectation**

```
>>> dico
{11: 'unu', 22: 'du', 33: 'tri'}
>>> dico[44]='kvar'
>>> dico
{11: 'unu', 22: 'du', 33: 'tri', 44: 'kvar'}
```

SHELL

- ▶ On peut supprimer un couple (clé:valeur) avec la commande **pop**

```
>>> dico.pop(33)
'tri'
>>> dico
{11: 'unu', 22: 'du', 44: 'kvar'}
```

SHELL

- ▶ On peut parcourir un dictionnaire en itérant sur les clés.

```
def parcours(dico):  
    for k in dico: # k est la clé  
        v = dico[k] # v est la valeur  
        print(f"{k} ({v})")
```

SCRIPT

```
>>> mousquetaires=dict()  
>>> mousquetaires["Athos"]=1615  
>>> mousquetaires["Porthos"]=1617  
>>> mousquetaires["Aramis"]=1620  
>>> mousquetaires["d'Artagnan"]=1615  
>>> parcours(mousquetaires)  
Athos (1615)  
Porthos (1617)  
Aramis (1620)  
d'Artagnan (1615)
```

SHELL

- 🍃 Partie I. Ensembles
- 🍃 Partie II. Fonctions de hachage
- 🍃 Partie III. Dictionnaires
- 🍃 **Partie IV. Mémoïsation**
- 🍃 Partie V. Compléments sur les chaînes
- 🍃 Partie VI. Systèmes de fichier
- 🍃 Partie VII. E/S : écrire dans un fichier
- 🍃 Partie VIII. E/S : lire dans un fichier

- ▶ Objectif : mémoriser les calculs déjà faits pour pouvoir les réutiliser.
- ▶ Exemple :
  - ▶ on calcule  $100!$  c'est « long », il faut 100 multiplications.
  - ▶ on calcule ensuite  $103!$  : il faut 103 multiplications.
  - ▶ Si on avait mémorisé  $100!$ , il aurait suffi de 3 multiplications.
  - ▶ car  $103! = 100! * 101 * 102 * 103$
- ▶ Il suffit de stocker les résultats dans un dictionnaire.
  - ▶  $n$  sera la clé
  - ▶ le résultat de  $\text{fact}(n)$  sera la valeur.
- ▶ Principe de l'algorithme
  - ▶ Si  $n$  est une clé du dictionnaire, on renvoie la valeur associée.
  - ▶ Sinon, on calcule  $v=n*\text{fact}(n-1)$  et on ajoute  $n:v$  dans le dictionnaire.
- ▶ Cette méthode s'appelle la **mémoïsation**.

```
mémoire_cache = { 0:1 } # fact(0)=1
```

SCRIPT

```
def fact(n):
    global mémoire_cache # global est facultatif
    # on ne modifie pas l'ensemble mais son contenu
    if n in mémoire_cache:
        return mémoire_cache[n]
    else:
        v = n*fact(n-1)
        mémoire_cache[n]=v
        return v
```

```
>>> from time import time
>>> len(mémoire_cache)
1
>>> t=time(); x=fact(800); t800=time()-t; len(mémoire_cache)
801
>>> t=time(); x=fact(810); t810=time()-t; len(mémoire_cache)
811
>>> print(t810/t800)
0.009458297506448839
>>> print(f'{t810/t800:.1%}')
0.9%
```

SHELL

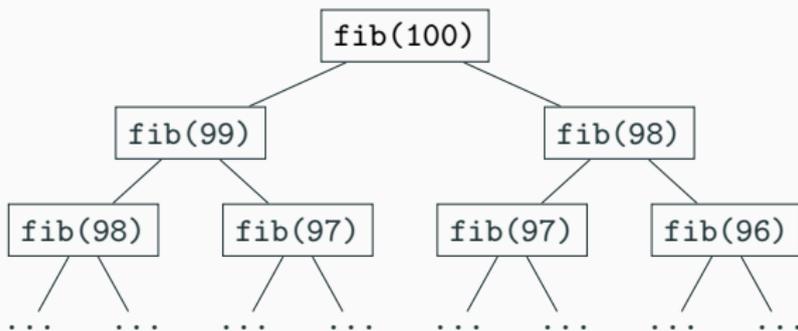
- Le calcul de fact(810) est 100 fois plus efficace que celui de fact(800)

- ▶ On a rencontré des récurrences doubles. Exemple : la suite de Fibonacci.
  - ▶ Très peu efficaces
  - ▶ Les mêmes calculs sont faits de nombreuses fois.

```
def fib(n):  
    if n < 2:  
        return 1  
    else:  
        return fib(n-1) + fib(n-2)
```

SCRIPT

On souhaite retenir les  
résultats intermédiaires  
(mémorisation)



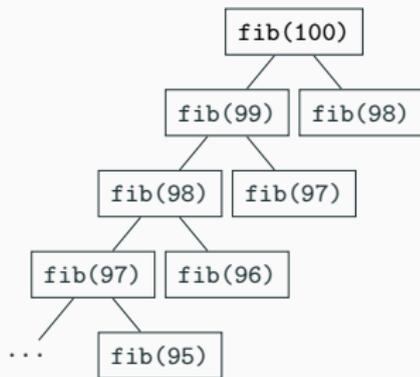
- ▶ Soit  $T_n$  le nombre d'appels récursif lors du calcul de `fib(n)` on a
  - ▶  $1 + T_n = 2 \cdot \text{fib}(n)$  car  $1 + T_n$  vérifie la même formule de récurrence que `fib(n)` (mais en partant de 2 au lieu de 1)

$$\begin{cases} T_n = 1 + T_{n-1} + T_{n-2} \\ T_0 = T_1 = 1 \end{cases} \quad \text{et donc} \quad \begin{cases} (1 + T_n) = (1 + T_{n-1}) + (1 + T_{n-2}) \\ (1 + T_0) = (1 + T_1) = 2 \end{cases}$$

- ▶ Pour calculer `fib(100)` il faut donc  $2 \cdot \text{fib}(100) - 1$  appels récursifs
  - ▶ Supposons que le calcul ne prenne que  $10^{-12}$  s par appel;
  - ▶ le temps nécessaire sera de  $\approx 10^9$  s  $\approx 36$  années.

<pre>mem = dict() def fib(n):     if n==0 or n==1:         mem[n] = 1     elif n not in mem:         mem[n] = fib(n-1)+fib(n-2)     return mem[n]</pre>	SCRIPT
<pre>&gt;&gt;&gt; fib(100) #Quasi instantané 573147844013817084101</pre>	SHELL

Le nouvel arbre d'appels est un « peigne » qui compte seulement 201 appels récursifs.



- Partie I. Ensembles
- Partie II. Fonctions de hachage
- Partie III. Dictionnaires
- Partie IV. Mémoïsation
- Partie v. Compléments sur les chaînes
- Partie VI. Systèmes de fichier
- Partie VII. E/S : écrire dans un fichier
- Partie VIII. E/S : lire dans un fichier

- ▶ On veut afficher le résultat d'une division euclidienne avec  $(n, d) = (17, 5)$

$$17 = 3 \times 5 + 2$$

- ▶ Version lourde : dur à modifier et à lire.

```
>>> str(n)+' = '+str(n//d)+' × '+str(d)+' + '+str(n%d)
'17 = 3 × 5 + 2'
```

SHELL

- ▶ Version plus lisible et plus aisément modifiable sur le long terme.

```
>>> f'{n} = {n//d} × {d} + {n%d}'
'17 = 3 × 5 + 2'
```

SHELL

L'ajout d'un **f** devant la chaîne permet d'utiliser des variables ou des expressions entre accolades. On parle de **f-string**.

- ▶ On peut choisir le nombre de chiffres après la virgule.

```
>>> pi
3.141592653589793
>>> f'{pi:.0f}    {pi:.2f}    {pi:.4f}'
'3      3.14    3.1416'
```

SHELL

- ▶ On peut préférer l'écriture scientifique ( $12345 = 1,2345 \times 10^4$ ) :

```
>>> a=12345
>>> f'{a:.0e}    {a:.1e}    {a:.2e}    {a:.6e}'
'1e+04    1.2e+04    1.23e+04    1.234500e+04'
```

SHELL

- ▶ On peut même utiliser des pourcentages :

```
>>> f'{0.1676:.2%}    {1/2:.0%}    {4/3:.4%}'
'16.76%    50%    133.3333%'
```

SHELL

- ▶ Le formatage permet aussi d'aligner des chaînes de tailles différentes.

```

texte = "Où suis-je ?"
print('#'*50) # Chaîne de 50 caractères
print(f"{texte:<50} fin") # aligné à gauche sur 50 caractères
print(f"{texte:>50} fin") # aligné à droite sur 50 caractères
print(f"{texte:^50} fin") # centré sur 50 caractères
print("")
print(f"{texte:*<50} fin")
print(f"{texte:*>50} fin") # On peut même préciser
print(f"{texte:*^50} fin") # les caractères à ajouter

```

SCRIPT

```

#####
Où suis-je ?                               fin
                                         Où suis-je ? fin
                        Où suis-je ?      fin

Où suis-je ?***** fin
*****Où suis-je ? fin
*****Où suis-je ?***** fin

```

SHELL

- ▶ Application : écrire sur 3 chiffres (à faire chez vous) :

```

1-11-21-31-41-51-61-71-81-91-101-avant
001-011-021-031-041-051-061-071-081-091-101-après

```

SHELL

- ▶ On peut stocker des informations dans une chaîne de caractères
  - ▶ avec les divers éléments **séparés par une virgule** (ou un espace).
  - ▶ On parle de format csv : utilisé par les tableurs
  - ▶ permet de stocker une feuille de calcul au format texte.
- ▶ On peut découper une chaîne pour récupérer la liste des éléments
  - ▶ méthode **split**

```
>>> 'mathématiques,12,10,15, 8,17'.split(',')
['mathématiques', '12', '10', '15', ' 8', '17']
>>> 'mathématiques 12 10 15 8 17'.split()
['mathématiques', '12', '10', '15', '8', '17']
```

SHELL

- ▶ Inversement, on peut recoller les éléments d'une liste de chaînes
  - ▶ méthode **sep.join(L)** (éléments de L séparés par sep)

```
>>> '-'.join(['partiel', '21', '03', '2024'])
'partiel-21-03-2024'
>>> ''.join(['partiel', '21', '03', '2024'])
'partiel21032024'
```

SHELL

- Partie I. Ensembles
- Partie II. Fonctions de hachage
- Partie III. Dictionnaires
- Partie IV. Mémoïsation
- Partie V. Compléments sur les chaînes
- Partie VI. Systèmes de fichier
- Partie VII. E/S : écrire dans un fichier
- Partie VIII. E/S : lire dans un fichier

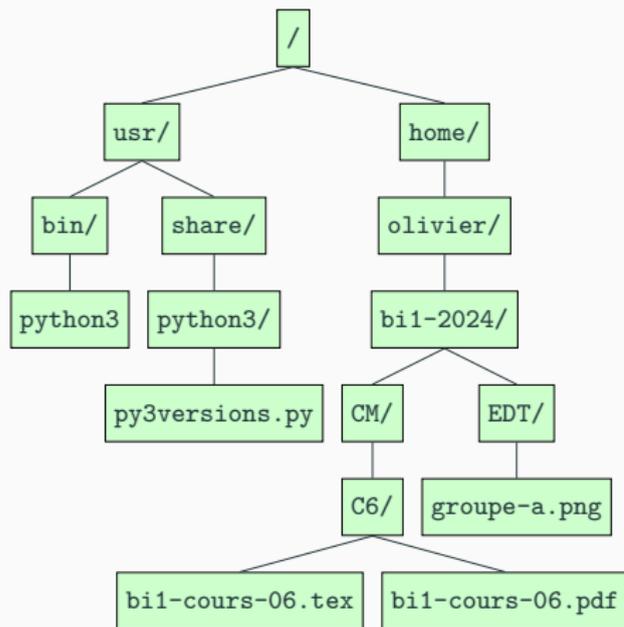
- ▶ Le mot **fichier** provient de **fiche** :  
*feuille de carton sur laquelle on écrit soit les titres des ouvrages que l'on veut cataloguer, soit les renseignements sur une personne ou un fait que l'on veut garder et retrouver facilement.*
- ▶ Le **fichier** désignait le recueil des fiches (ou le meuble les contenant).
- ▶ Un **fichier informatique** est un ensemble structuré d'information
  - tableau : feuille de calcul
  - texte (txt, html, odt).
  - suite d'instructions : programme
  - données quelconques (image, vidéo, etc.)
- ▶ Les **systèmes d'exploitation** permettent de travailler sur des fichiers
  - ▶ stockage de grandes quantités d'informations
  - ▶ recherche, classement, modification
  - ▶ fichiers stockés sur le disque dur (ou clé USB, etc)
- ▶ Système d'exploitation : *Operating System* (OS).
  - ▶ exemples : GNU/Linux, Android, Windows, macOS, etc

- ▶ Le père de tous les systèmes d'exploitation moderne est **UNIX** ( $\approx$  1975).
- ▶ Il est à l'origine de tous les systèmes d'exploitation digne de ce nom.
  - Pour les serveurs :
    - ▶ La famille **BSD** : FreeBSD, NetBSD, OpenBSD (sécurité), dragonflyBSD
    - ▶ La famille propriétaire : AIX (IBM), Solaris (Oracle), HP-UX (HP),...
    - ▶ La famille **GNU/Linux** : Debian, Ubuntu, Red Hat, Gentoo,...
  - Pour les particuliers :
    - ▶ macOS (anciennement Mac OS X)
    - ▶ La famille GNU/Linux : **Debian**, Ubuntu, Red Hat, Gentoo,...
    - ▶ Les autres Linux (android)
- ▶ L'autre grande famille est composée des **Windows**...

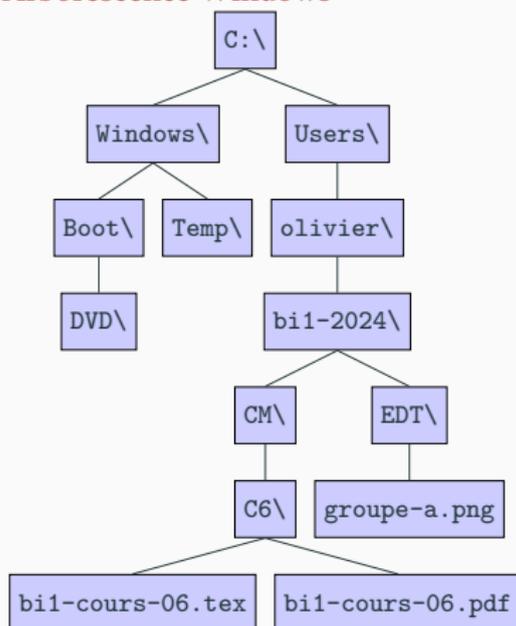
- ▶ Les fichiers sont regroupés dans une **arborescence**,
  - ▶ avec une ou plusieurs **racines**
  - ▶ les nœuds sont les **répertoires** (ou dossiers)
  - ▶ les feuilles sont les **fichiers**
  - ▶ un répertoire peut contenir des sous-répertoires et des fichiers.
- ▶ Il y a essentiellement deux systèmes d'arborescence :
  - ▶ **Unix** (GNU/Linux, macOS, BSD) avec des petites variantes
  - ▶ **Windows**
- ▶ Sur Unix, une seule racine (nommée **/**).
- ▶ Sur Windows, plusieurs racines (nommées **C:\**, **D:\**, etc).
  - ▶ En général, **C:\** représente le disque principal.

- ▶ Un répertoire est relié à la racine par un chemin

## Arborescence Unix



## Arborescence Windows



- ▶ Chemin :

- ▶ **Unix** `/home/olivier/bi1-2024/EDT/groupe-a.png`
- ▶ **Windows** `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`

- ▶ Il y a deux façons d'indiquer où se trouve un fichier ou un répertoire
  - ▶ Les chemins absolus
  - ▶ Les chemins relatifs
- ▶ **Chemins absolus** : chemin complet à partir de la racine.
  - ▶ `/home/olivier/bi1-2024/EDT/groupe-a.png`
  - ▶ `C:\Users\olivier\bi1-2024\EDT\groupe-a.png`
- ▶ **Chemins relatifs** : chemin à partir du répertoire courant.
  - `EDT/groupe-a.png` peut correspondre à
    - ▶ `/home/olivier/bi1-2024/EDT/groupe-a.png`
    - ▶ `/home/olivier/bi1-2024/sauvegarde/EDT/groupe-a.png`
  - Cela dépend si je suis dans sauvegarde ou bi1-2024
- ▶ Le répertoire parent (du dessus) se note `..`
  - ▶ Si je suis dans `/home/olivier`, `..` est le chemin vers `/home`
  - ▶ La racine `/` est le seul répertoire sans parent. `:'(`

► Le module `os` permet d'utiliser des équivalents aux commandes Unix

- `getcwd()` renvoie le répertoire courant (commande Unix : `pwd`)

```
>>> import os
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8'
```

SHELL

- `chdir(...)` permet de changer de répertoire (Unix : `cd`)

```
>>> os.chdir("dossier/") # Chemin relatif
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'

>>> os.chdir('/home/olivier/bi1-2024/CM/C8/dossier') #Chemin absolu
>>> os.getcwd()
'/home/olivier/bi1-2024/CM/C8/dossier'
```

SHELL

- `listdir()` renvoie les fichiers du répertoire (Unix : `ls`)

```
>>> os.listdir()
['exo2.py', 'exo1.py', 'sujet.pdf']
```

SHELL

- ▶ Un chemin peut être codé en Python par une chaîne.
- ▶ Problème avec Windows (qui ne suit pas les conventions UNIX) :
  - ▶ les \ doivent être doublés
  - ▶ \ est un caractère d'échappement dans une chaîne ("`\n`" "`\"` "`\\` ")
  - ▶ Linux : `'/home/olivier/bi1-2024/EDT/groupe-a.png'`
  - ▶ Windows : `'C:\\home\\olivier\\bi1-2024\\EDT\\groupe-a.png'`
- ▶ Un bon logiciel doit fonctionner sur tous les OS.
- ▶ On peut demander en Python sur quel système on travaille.

```
>>> os.name #Sous Linux SHELL  
'posix'
```

```
>>> os.name #Sous Windows SHELL  
'nt'
```

- ▶ On peut construire un chemin de manière portable.

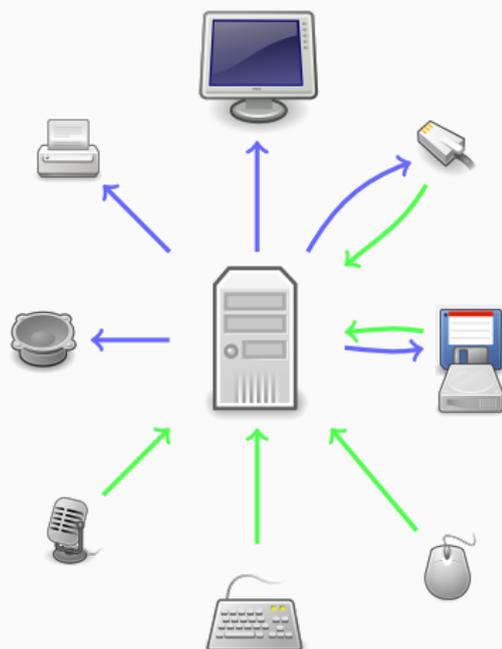
```
>>> os.path.join('.', 'QCM', 'q1.xml') # Sous Linux SHELL  
'../QCM/q1.xml'
```

```
>>> os.path.join('.', 'QCM', 'q1.xml') # Sous Windows SHELL  
'..\\QCM\\q1.xml'
```

<code>os.getcwd()</code>	renvoie le répertoire courant
<code>os.chdir(path)</code>	change de répertoire courant
<code>os.listdir(path='.')</code>	liste des fichiers et répertoires
<code>os.path.join(path1,path2,...)</code>	construction portable d'un chemin
<code>os.remove(path)</code>	suppression d'un fichier
<code>os.path.isfile(path)</code>	test d'existence d'un fichier
<code>os.path.isdir(path)</code>	test d'existence d'un répertoire
<code>os.path.split(path)</code>	pour extraire le fichier d'un chemin
<code>os.path.getsize(path)</code>	la taille d'un fichier

<https://docs.python.org/fr/3.7/library/os.html>

- 🍃 Partie I. Ensembles
- 🍃 Partie II. Fonctions de hachage
- 🍃 Partie III. Dictionnaires
- 🍃 Partie IV. Mémoïsation
- 🍃 Partie V. Compléments sur les chaînes
- 🍃 Partie VI. Systèmes de fichier
- 🍃 **Partie VII. E/S : écrire dans un fichier**
- 🍃 Partie VIII. E/S : lire dans un fichier



L'ordinateur communique avec le reste du monde

▶ Entrée :

- ▶ Souris et clavier
- ▶ Micro, caméra
- ▶ Disque dur, clé usb, disquette
- ▶ Câble ethernet (internet)

▶ Sortie :

- ▶ Écran
- ▶ Imprimante
- ▶ Disque dur, clé usb, disquette
- ▶ Enceinte

- ▶ On souhaite créer un fichier `test.txt` contenant des résultats.
- ▶ Nous ne travaillerons dans ce cours qu'avec des fichiers de texte.
- ▶ Dans un tel fichier, nous déposerons des chaînes de caractères. Pour y écrire `-234`, nous déposerons `'-234'`.
- ▶ Commençons par ouvrir un fichier `test.txt` en écriture (`write`).

```
fichier = open('test.txt', 'w', encoding = 'utf-8')
```

SCRIPT

- ▶ On peut indiquer un chemin (absolu ou relatif) menant au fichier.
  - ▶ Si un ancien fichier de ce nom existe, il sera remplacé.

```
>>> fichier  
<_io.TextIOWrapper name='test.txt' mode='w' encoding='utf-8'>
```

SHELL

- ▶ La valeur de `fichier`, renvoyé par `open`, est un **descripteur de fichier**.
- ▶ L'encodage par défaut dépend de la machine.

- ▶ Une fois le fichier ouvert, il est prêt à recevoir des données.
  - ▶ si le fichier n'existait pas, on va le créer.
  - ▶ s'il existait déjà : on l'écrase et on recommence à zéro.
- ▶ On écrit dans le fichier avec la méthode `write` du descripteur de fichier.

```
fichier.write('Bonjour tout le monde !\n')  
fichier.write('Voici un texte écrit depuis Python !\n')
```

SCRIPT

- ▶ Il faut déposer le caractère de retour à la ligne `'\n'`,
  - ▶ sinon, le prochain `write` prendra effet sur la même ligne.
- ▶ Les écritures sont mises en tampon ; elles ne prennent pas forcément effet immédiatement. À la fin du traitement, **il faut fermer le fichier** (`close`) pour que tout soit bien écrit.

```
fichier.close()
```

SCRIPT

- ▶ Dans le fichier, on ne peut écrire que des chaînes de caractères.

```
>>> fichier = open('test.txt', 'w', encoding = 'utf-8')
>>> n=1991
>>> fichier.write(n)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: write() argument must be str, not int
```

SHELL

- ▶ Pour écrire un nombre, il faut d'abord le convertir en chaîne

```
fichier.write(str(n))
fichier.write(" est l'année de naissance de Python.\n")
```

SCRIPT

- ▶ ou encore...

```
fichier.write(f"{n} est l'année de naissance de Python.\n")
```

SCRIPT

SCRIPT

```
def créer_table(n):  
    """crée un fichier contenant la table  
    de multiplication de n"""  
    # étape 1 : ouverture du fichier  
    fichier=open('table'+str(n)+'.txt','w',encoding='utf-8')  
    # étape 2 : écriture dans le fichier  
    for i in range(1, 11):  
        fichier.write(str(i)+'*'+ str(n)+'='+str(i*n)+'\n')  
    # étape 3 : fermeture du fichier  
    fichier.close()  
  
créer_table(7)
```

- Pour lire le fichier table7.txt, on utilise un lecteur de fichier texte.

TABLE7.TXT

```
1*7=7  
2*7=14  
3*7=21  
4*7=28  
5*7=35  
6*7=42  
7*7=49  
8*7=56  
9*7=63  
10*7=70
```

SCRIPT

```
def créer_table(n):  
    """crée un fichier contenant la table  
    de multiplication de n"""  
    # étape 1 : ouverture du fichier  
    fichier = open(f'table{n}.txt','w',encoding='utf-8')  
    # étape 2 : écriture dans le fichier  
    for i in range(1, 11):  
        fichier.write(f'{i:>2} × {n} = {i*n:>2}\n')  
    # étape 3 : fermeture du fichier  
    fichier.close()  
  
créer_table(7)
```

- Les nombres sont correctement alignés.

TABLE7.TXT

```
1 × 7 = 7  
2 × 7 = 14  
3 × 7 = 21  
4 × 7 = 28  
5 × 7 = 35  
6 × 7 = 42  
7 × 7 = 49  
8 × 7 = 56  
9 × 7 = 63  
10 × 7 = 70
```

- ▶ Il peut être intéressant d'ajouter des lignes à un fichier. Il faut alors l'ouvrir en écriture en mode 'a' (ajout) et non 'w'.

```
fichier = open('test.txt', 'a', encoding = 'utf-8')
```

SCRIPT

- ▶ Pour ajouter deux lignes à la fin du fichier table7.txt.

```
def allonger_fichier(n):  
    fichier = open(f'table{n}.txt', 'a', encoding='utf-8')  
    for i in range(11, 13):  
        fichier.write(f'{i:>2} × {n} = {i*n:>2}\n')  
    fichier.close()
```

SCRIPT

- ▶ Remarque : il n'est pas possible de supprimer des lignes dans un fichier directement (mais on peut créer un nouveau fichier et détruire l'ancien).

- 🍃 Partie I. Ensembles
- 🍃 Partie II. Fonctions de hachage
- 🍃 Partie III. Dictionnaires
- 🍃 Partie IV. Mémoïsation
- 🍃 Partie V. Compléments sur les chaînes
- 🍃 Partie VI. Systèmes de fichier
- 🍃 Partie VII. E/S : écrire dans un fichier
- 🍃 Partie VIII. E/S : lire dans un fichier

- ▶ Problème inverse : comment lire (**read**) le fichier `table5.txt` ?
- ▶ On doit connaître son encodage; on sait qu'il est en `utf-8`.

```
>>> fichier = open('table5.txt', 'r', encoding='utf-8') SHELL
```

- ▶ On peut lire d'un seul coup **la totalité du fichier** dans une seule chaîne
  - ▶ avec la méthode **read()**.

```
>>> texte = fichier.read() SHELL
>>> fichier.close()
>>> texte # contient tout le fichier !
'5 x 1 = 5\n5 x 2 = 10\n5 x 3 = 15\n5 x 4 = 20\n'
>>> print(texte)
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
```

- ▶ On peut itérer directement sur les lignes avec une boucle `for`

```
def affiche_et_compte(f):  
    i=0 # pour compter le nombre de lignes  
    fichier = open(f, 'r', encoding='utf-8')  
    for ligne in fichier:  
        print('>' ,ligne, end = '')  
        i=i+1  
    fichier.close()  
    return i # nombre de lignes
```

SCRIPT

```
>>> affiche_et_compte('table5.txt')  
> 5 × 1 = 5  
> 5 × 2 = 10  
> 5 × 3 = 15  
> 5 × 4 = 20  
4
```

SHELL

- ▶ Une méthode classique de travail sur fichier texte
  - ▶ on prend un fichier en entrée
  - ▶ on produit un autre fichier en sortie
- ▶ Exemple :
  - ▶ lecture d'un fichier : chaque ligne contient des nombres
  - ▶ écriture d'un fichier : chaque ligne contient la somme de ces nombres.

83 21 10 34 98	ALÉA.TXT
11 34 18 69 76	
92 88 28 54 62	
17 33 45 13 82	
18 11 88 67 85	
88 51 89 66 20	
98 49 72 29 59	
33 78 86 42 62	

246	SOMME.TXT
208	
324	
190	
269	
314	
307	
301	

- Pour générer le fichier aléatoire, on doit écrire 8 lignes de 5 colonnes chacune ; on utilise deux boucles imbriquées.

```
def créer_aléa(nom_fichier):  
    fichier = open(nom_fichier, 'w', encoding = 'utf-8')  
    for ligne in range(8): # je produis 8 lignes  
        for colonne in range(5): # chaque ligne a 5 colonnes  
            fichier.write(f'{randint(10,99)} ')  
            fichier.write('\n') # je vais à la ligne  
    fichier.close()
```

SCRIPT

```
>>> créer_aléa('aléa.txt')
```

SHELL

					ALÉA.TXT
83	21	10	34	98	
11	34	18	69	76	
92	88	28	54	62	
17	33	45	13	82	
18	11	88	67	85	
88	51	89	66	20	
98	49	72	29	59	
33	78	86	42	62	

- ▶ Pour générer le fichier somme.txt,
  - ▶ on lit aléa.txt ligne par ligne
  - ▶ à chaque ligne lue, on écrit la somme de la ligne dans somme.txt

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée = open('aléa.txt', 'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # je calcule la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SCRIPT

```
83 21 10 34 98
11 34 18 69 76
92 88 28 54 62
17 33 45 13 82
18 11 88 67 85
88 51 89 66 20
98 49 72 29 59
33 78 86 42 62
```

ALÉA.TXT

```
246
208
324
190
269
314
307
301
```

SOMME.TXT

- ▶ Pour ceux qui ronflent au fond de l'amphi.
  - ▶ Pour comprendre ce que fait un programme...
  - ▶ ... on arrête de se tourner les pouces et on sort sa console Python!

```
sortie = open('somme.txt', 'w', encoding = 'utf-8')
entrée = open('aléa.txt', 'r', encoding = 'utf-8')

for ligne in entrée: # j'itère sur les lignes de entrée
    L1 = ligne.split() # je récupère les entiers sur la ligne
    L2 = [int(x) for x in L1] # je les convertis en entiers
    S2 = sum(L2) # Je calcul la somme dans S2
    sortie.write(f'{S2}\n') # j'écris S2 dans somme.txt

entrée.close() ; sortie.close()
```

SCRIPT

```
>>> '83 21 10 34 98\n'.split() # L1
['83', '21', '10', '34', '98']
>>> [int(x) for x in ['83', '21', '10', '34', '98']] # L2
[83, 21, 10, 34, 98]
>>> sum([83,21,10,34,98]) # S2
246
>>> f'{246}\n' # Ce qu'on écrit dans le fichier
'246\n'
```

SHELL

- ▶ Si le fichier n'existe pas.

```
>>> fichier = open('complot.py', 'r', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'complot.py'
```

SHELL

- ▶ S'il n'y a pas les permissions pour modifier le fichier.

```
>>> fichier = open('lecture_seule.txt', 'w', encoding='utf-8')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
PermissionError: [Errno 13] Permission denied: 'lecture_seule.txt'
```

SHELL

- ▶ Si le descripteur de fichier a été fermé.

```
>>> fichier = open('table5.txt', 'r', encoding = 'utf-8')
>>> fichier.close()
>>> fichier.read()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: I/O operation on closed file.
```

SHELL

- ▶ etc.

- ▶ Ouverture en lecture (read) :

```
fichier = open('fichier.txt', 'r', encoding='utf-8')
```

SCRIPT

- ▶ Ouverture en écriture (write ou add) :

```
fichier = open('fichier.txt', 'w', encoding='utf-8')
```

SCRIPT

ou

```
fichier = open('fichier.txt', 'a', encoding='utf-8')
```

SCRIPT

- ▶ Lecture :

- ▶ fichier.read()
- ▶ for ligne in fichier:

- ▶ Écriture :

- ▶ fichier.write(chaine)

- ▶ Fermeture :

```
fichier.close()
```

SCRIPT

Merci pour votre attention

Questions



# Cours 8 — Ensembles, dictionnaires et fichiers texte

## Partie I. Ensembles

Ensembles Python  
Parcours d'un ensemble  
Appartenance et inclusion  
Nombre d'éléments d'un ensemble  
Construire un ensemble  
Modifier un ensemble  
Opérations sur les ensembles  
Que puis-je mettre dans un ensemble?

## Partie II. Fonctions de hachage

Qu'est-ce?  
Pourquoi?  
Application aux ensembles  
Résumé

## Partie III. Dictionnaires

Qu'est-ce?  
Accès aux éléments  
Exceptions  
Modifier un dictionnaire

Parcours d'un dictionnaire

## Partie IV. Mémoïsation

Principes : exemple de la factorielle  
Implémentation de la factorielle  
Mémoïsation et Fibonacci  
Implémentation de Fibonacci

## Partie V. Compléments sur les chaînes

Chaînes de formatage  
Formatage avancé : flottant  
Formatage avancé : alignement  
Découper une ligne de texte

## Partie VI. Systèmes de fichier

Utilité des fichiers  
Un peu d'histoire  
Système de fichiers  
Arborescence  
Chemins relatifs et absolus  
Unix et le module os  
Chemins et chaînes de caractères

Quelques fonctions utiles du module os

## Partie VII. E/S : écrire dans un fichier

Entrées/Sorties (E/S)  
Ouverture d'un fichier en écriture  
Écriture dans un fichier  
Formater les écritures  
Exemple : générer une table de multiplication

Exemple : générer une jolie table de multiplication

Écrire à la fin d'un fichier

## Partie VIII. E/S : lire dans un fichier

Lecture d'un fichier : `read`  
Lecture d'un fichier : `for`  
Lire, écrire, compter : objectifs  
Lire, écrire, compter : construire `aléa.txt`  
Lire, écrire, compter : générer `somme.txt`  
Lire, écrire, compter : explications  
Les exceptions liées aux fichiers  
Résumé sur les fichiers texte

## Partie IX. Table des matières

- ▶ © 2024 — Olivier Baldellon
- ▶ Ce document est publié sous licence **CC-BY Attribution 4.0** 
- Vous êtes autorisé à :
  - ▶ **Partager** — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats pour toute utilisation, y compris commerciale.
  - ▶ **Adapter** — remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.
- Selon les conditions suivantes :
  - ▶ **Attribution** — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.
- ▶ <https://creativecommons.org/licenses/by/4.0/deed.fr>