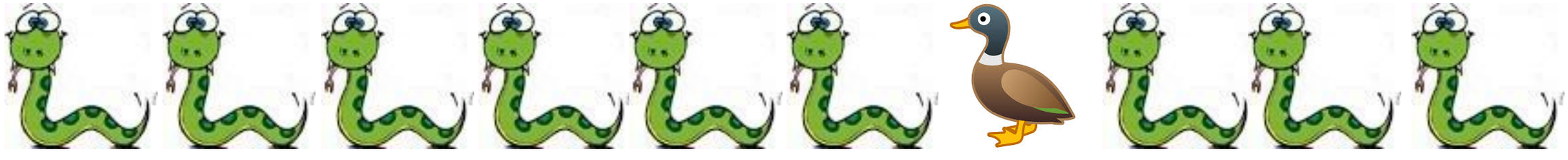
Université Nice Sophia Antipolis
Faculté des Sciences

Programmation avec Python — 2019
Etienne Lozes (remerciements : Jean-Paul Roy)
L1-Sciences

Cours n°6

http://deptinfo.unice.fr/~elozes

Fichiers et exceptions



1 . Les exceptions

Les phénomènes exceptionnels

• Vous savez qu'il peut y avoir des **erreurs** dans un calcul, dues à la mauvaise gestion des cas particuliers :

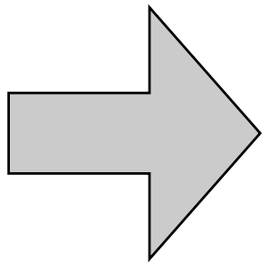
```
>>> L = [6, 4, 7, 2, 1, 8]
>>> L[6]
IndexError: list index out of range
```

```
>>> L.index(5)
ValueError: 5 is not in list
```

```
>>> 2 / 0
ZeroDivisionError: division by zero
```

- **IndexError**, **ValueError**, etc sont des **exceptions**.
- Si l'on ne fait rien, le programme s'arrête brutalement et affiche **la trace d'exécution,** i.e. le contexte dans lequel l'exception **a été levée**

```
1 def f(x) :
2     return g(x) + 1
3
4 def g(x) :
5     return 1 / x
6
7 f(0)
```



```
Traceback (most recent call last):
  File "toto.py", line 7
    f(0)
  File "toto.py", line 2, in f
    return g(x) + 1
  File "toto.py", line 5, in g
    return 1 / x
ZeroDivisionError: division by zero
```

Eviter les exceptions?

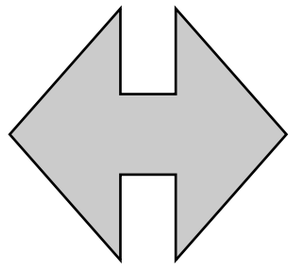
Les exceptions ne sont pas forcément des erreurs de programmation; elles peuvent être dues à des informations incorrectes fournies par l'utilisateur…

```
def record_cent_metre(chrono) :
    if chrono < 9.58 :
        return True
    return False
```

```
>>> record_cent_metre('2 sec')
TypeError:'<' not supported between
instances of 'str' and ‘float'
```

On peut vérifier les informations et appeler les instructions « à risque » une fois qu'on est sûr de ne pas faire d'erreur …

```
def record_cent_metre(chrono) :
    if type(chrono) == float:
        if chrono < 9.58 :
            return True
    return False
```



```
def record_cent_metre(chrono) :
    if type(chrono) == float and chrono < 9.58 :
        return True
    return False
```

et paresseux, cf cours 1

… mais il est facile d'oublier des cas!

```
>>> record_cent_metre(2)
False
```

```
>>> record_cent_metre(2.0)
True
```

```
>>> type(2) == float
False
```

Rattraper les exceptions!

Au lieu de chercher à éviter les exceptions, on peut chercher à les traiter à part, dans une partie du programme dédiée aux cas exceptionnels

```
def record_cent_metre(chrono) :
    try :
        if chrono < 9.58 :
            return True
        return False
    except :
        return False
```

```
try :
    <instr1>
    ...
except :
    <instr2>
    ...
```

- Si au cours de l'exécution du bloc d'instructions <*instr*$_1$>... une exception se produit, l'exécution du bloc est abandonnée et le bloc <*instr*$_2$>... est exécuté.
- Si l'exécution du bloc <*instr*$_1$>... s'est déroulée normalement, le bloc except n'est pas utilisé.

```
1 def record_cent_metre(chrono) :
2     try :
3         if chrono < 9.58 :
4             return True
5         return False
6     except :
7         return False
8
9 record_cent_metre(2)
```

Exercice : tester cette fonction avec chrono=2, 10.5, et '2' à l'aide de Python tutor

La fonction chercher(x,L) ci - dessous retourne la position de la première occurrence de x dans L, ou bien -1.

```
def chercher(x, L):
  for i in range(len(L)):
    if L[i] == x :
       return i
  return -1
```

```
>>> L = [3,2,8,6,2,4,7,6,5]
>>> chercher(6,L)
3
```

• Or Python fournit la méthode index qui demande à une liste L la position d'un élément. Hélas elle ne renvoie pas -1 mais **lève une exception** en cas d'échec ! Comment transformer cette erreur en -1 ?

```
>>> L.index(6)
3
>>> L.index(9)
ValueError
```

• Réponse : en traitant l'exception, en essayant de la capturer au passage pour proposer un traitement de ce cas exceptionnel.

• Retenez : **il n'y a pas d'erreur, seulement des EXCEPTIONS !**

- Si j'invoque brutalement la méthode index en demandant `L.index(x)`, je risque de provoquer une exception.

- Je vais simplement **ESSAYER** de faire le calcul et regarder ensuite si j'ai fait une erreur...

```
def cherche(x,L) :
    try :
        res =  L.index(x)
        # pas d'erreur? on continue
        return res
    except :
        # une erreur? on reprend ici
        return -1
```

Autre exemple. Je souhaite écrire la fonction moyenne(L) qui renvoie la moyenne de la liste L, et qui renvoie -1 si la liste est vide

```
def moyenne(L) :
    try :
        res =  sum(L) / len(L)
        # pas d'erreur? on continue
        return res
    except ZeroDivisionError :
        # une erreur de division par 0? on reprend ici
        return -1
```

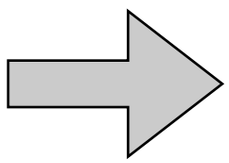
- Il y a différentes exceptions : `ValueError`, `ZeroDivisionError`, `KeyError`, etc. Un bloc `except:` les attrape toutes. Il est possible de n'attraper que certaines exceptions, ou d'envisager des traitements spécialisés pour chaque exception possible...

```
>>> moyenne([])
-1
>>> moyenne(['toto'])
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Autre exemple : reconnaitre un nombre dans une chaîne

J'ai une solution moyennement satisfaisante avec la méthode isdigit.

```
def est_nombre(s) :
    return s.isdigit()
```



```
>>> est_nombre('2')
True
>>> est_nombre('3.14')
False
```

À la place, je peux tenter de convertir la chaîne de caractères en flottant, juste pour voir si ça provoque une erreur…

```
def est_nombre(s) :
    try :
        x = float(s)
        return True
    except :
        return False
```

Comment lever une exception?

C'est une bonne pratique de lever une exception volontairement dans certains cas. Cela permet d'avoir des messages d'erreurs plus lisibles et de corriger son code plus facilement.

```
def moyenne(L) :
    assert( len(L) != 0 )
    # lève une exception si len(L) == 0
    return sum(L) / len(L)
```

```
>>> moyenne([])
…
assert(len(L) != 0) line 2 in f
…
AssertionError
```

Méthode 1 : assert

```
def moyenne(L) :
    try :
       return  sum(L) / len(L)
   except ZeroDivisionError :
        # une erreur de division par 0?
        # on la remplace par une erreur plus explicite
        raise ValueError('moyenne(L): L doit être non vide')
```

```
>>> moyenne([])
…
ValueError :
moyenne(L) : L doit
être non vide
```

Méthode 2 : raise

La méthode raise est plus générale, elle permet de lever n'importe quelle exception.

On peut même créer ses propres exceptions (mais avec des objets, donc hors programme).

Exceptions et « return longue distance »

```
def f() :
    # déclenche une exception mais ne la rattrape pas
    2 / 0
    print('ce message ne sera pas affiché')

def g() :
    f() # déclenche une exception
    print('ce message ne sera pas affiché')

def h()
    try :
       g()
    except :
       pass  # <- ne fait rien, compte pour un bloc vide
    print('ce message sera affiché’)

h()
```

Dans cet exemple, l'exception interrompt la fonction f, mais aussi la fonction g qui avait appelé la fonction f: on passe directement de la ligne 2/0 à la ligne pass. (Exercice: vérifier avec Python tutor)

2. Les fichiers

Utilité des fichiers

- Le mot **fichier** provient du terme 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 qui contient les fiches.

- Les **systèmes d'exploitation** (Linux, MacOS, Windows) permettent de stocker de grandes quantités d'informations, de les rechercher et de les classer sur disques durs (situés où ?) dans des fichiers un peu comme les fiches cartonnées.

- Système d'exploitation : ***Operating system*** (**OS**). La référence pour un informaticien est **UNIX** (Linux, MacOS).

Il y a deux systèmes de fichiers possibles

• Il y a essentiellement deux déclinaisons de systèmes de fichiers (*filesystems*) : **Unix** (Linux, MacOS) et **Windows**.

• Avec quelques légères différences entre **Linux** (Unix System V) et **MacOS** (Unix BSD).

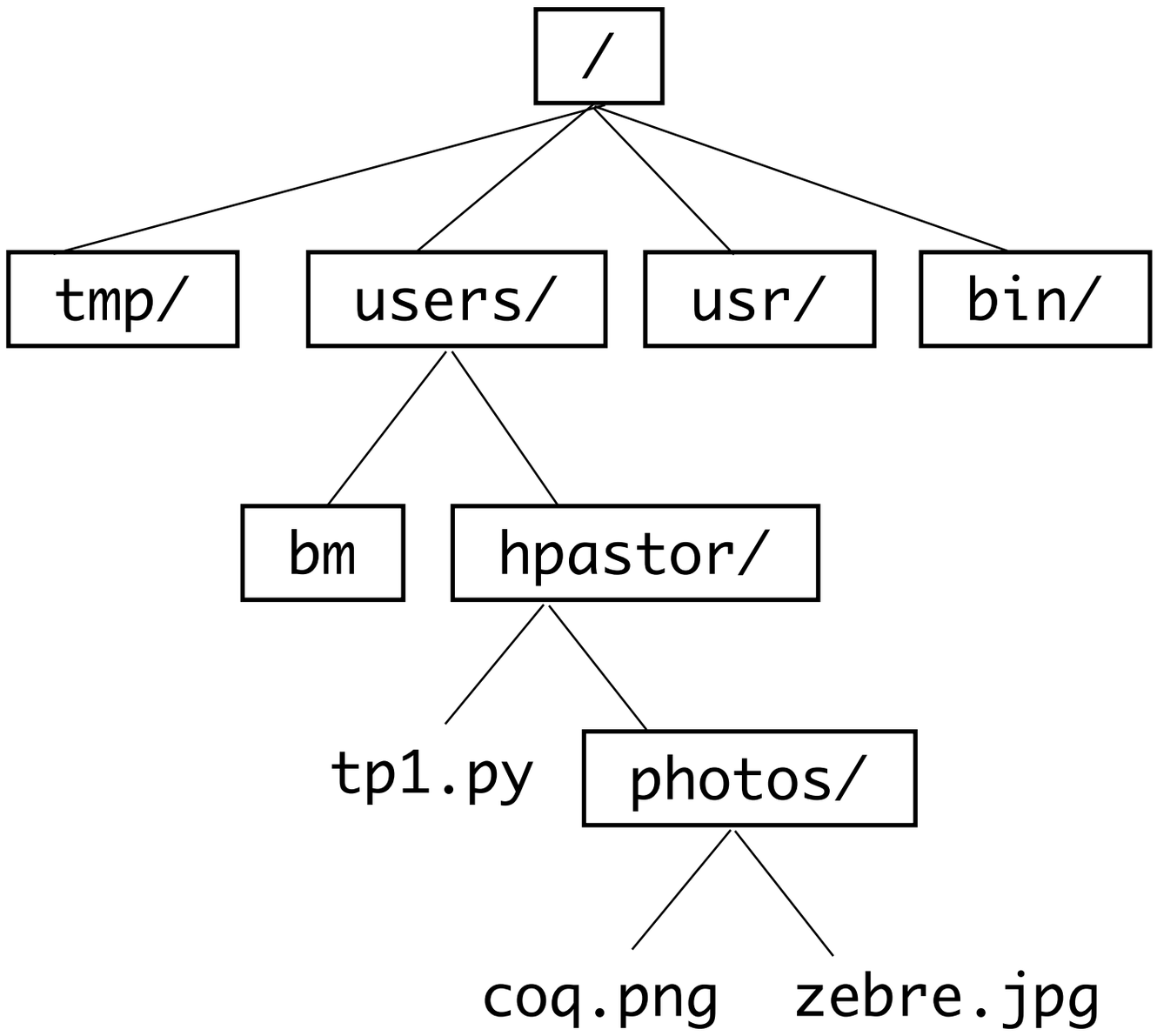
• Les fichiers (*files*) sont regroupés dans une **arborescence** (*tree*) ayant une ou plusieurs **racines** (*roots*), dont les noeuds sont les **répertoires** (*directories, folders*) et les feuilles sont les **fichiers** (*files*).

• Un répertoire contient des répertoires et des fichiers.

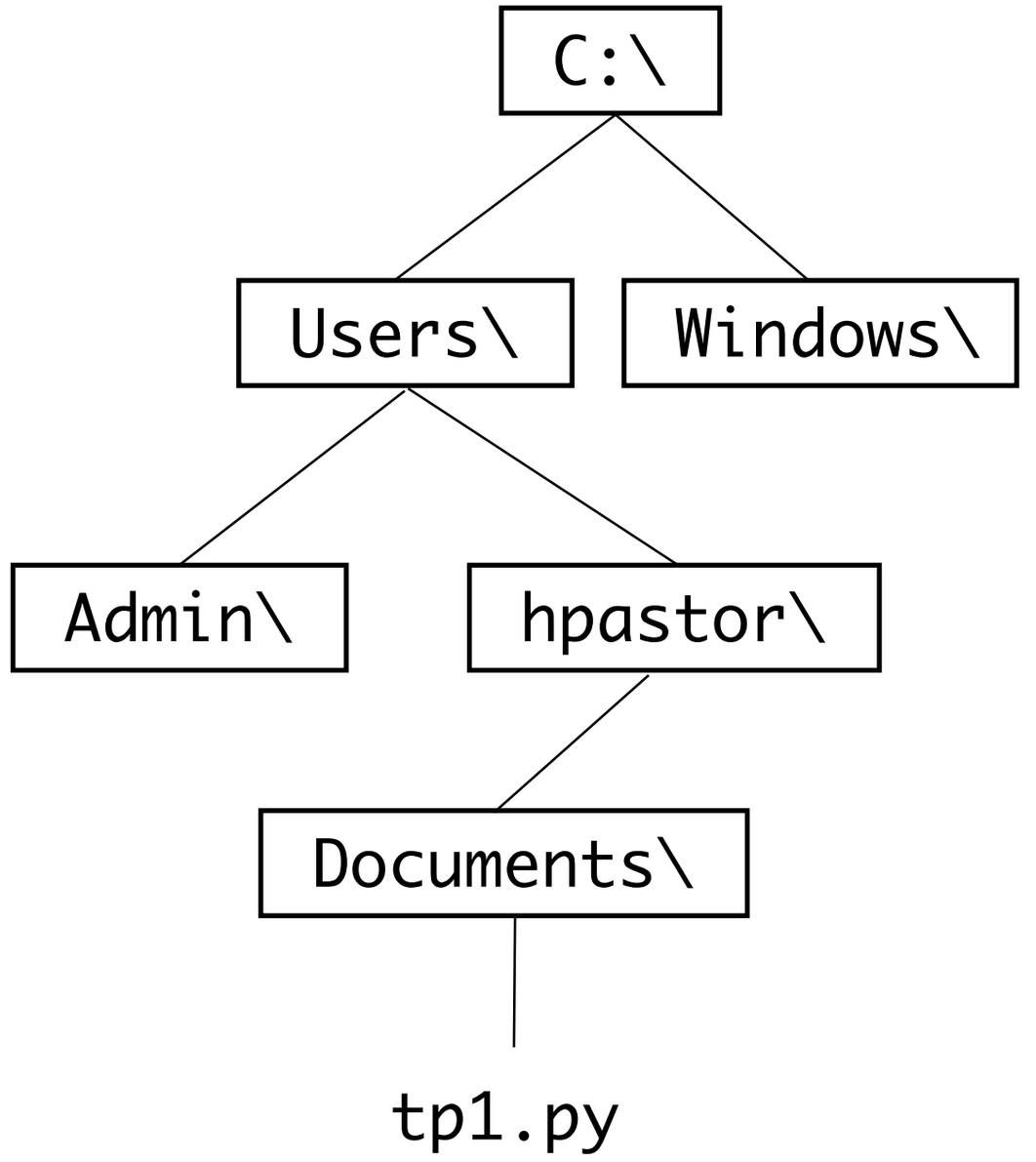
• Sur **Unix**, une seule racine nommée `/`

• Sur **Windows**, plusieurs racines nommées `A:\`, `B:\`, `C:\`, etc. En général `C:\` représente le disque principal.

UNIX (Linux)



WINDOWS



- Un répertoire ou un fichier est relié à la racine par un **chemin**.

```
/users/hpastor/                          c:\Users\hpastor\
/users/hpastor/photos/coq.png            c:\Users\hpastor\Documents\tp1.py
```

Chemins relatifs et absolus

• Les **chemins absolus** spécifient l'emplacement exact d'un répertoire ou fichier à l'intérieur d'une arborescence, à partir de la racine.

(Linux) `/users/hpastor/photos/coq.png`

(MacOS) `/Volumes/macleUSB/Python/tp1.py`

(Windows) `c:\Users\hpastor\Documents\tp1.py`

• Les **chemins relatifs** spécifient l'emplacement d'un répertoire ou fichier à partir d'un certain répertoire de l'arborescence (implicite, non indiqué dans le chemin lui-même).

Ex : `photos/coq.png` *est un chemin relatif à* `/users/hpastor/`

• Le répertoire père se note `..`

Ex : `../../bm` *si je suis dans le répertoire* `photos`

Le répertoire courant et le module `os`

• Il est important de savoir dans quel répertoire on est en train de travailler, afin d'accéder aux fichiers par des chemins relatifs. Ce répertoire de travail est le **répertoire courant** (*current directory*).

• Le module os de Python permet de manipuler le système de fichiers sans sortir de Python (au toplevel ou dans une fonction).

```
import os
```

• Je peux demander quel est le répertoire courant si je suis perdu :

```
>>> os.getcwd()
'/Users/jpr/Documents'
```

Get Current Working Directory

• Je peux changer de répertoire courant et me déplacer (de façon relative ou absolue) vers un autre répertoire de l'arborescence :

```
>>> os.chdir('../Desktop/Exemples/')
```

Change Directory

Chemins et chaînes de caractères

- Un chemin peut être codé en Python par une chaîne. ***Problème avec Windows qui ne suit pas les conventions UNIX*** : les \ doivent être doublés (puisque \ est un caractère d'échappement dans une chaîne).

(Linux/Mac) `'/users/hpastor/photos/coq.png'`

(Windows) `'c:\\Users\\hpastor\\Documents\\tp1.py'`

- **Un bon logiciel doit fonctionner sur tous les OS**. On peut demander en Python sur quel système on travaille :

```
>>> os.name
'posix'
```
Linux/Mac

```
>>> os.name
'nt'
```
Windows

- On peut construire un chemin de manière portable :

```
>>> os.path.join('hpastor','photos','coq.png')
'hpastor/photos/coq.png'
```
Linux/Mac

```
>>> os.path.join('hpastor','Documents','tp1.py')
'hpastor\\Documents\\tp1.py'
```
Windows

Quelques fonctions utiles du module `os`

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

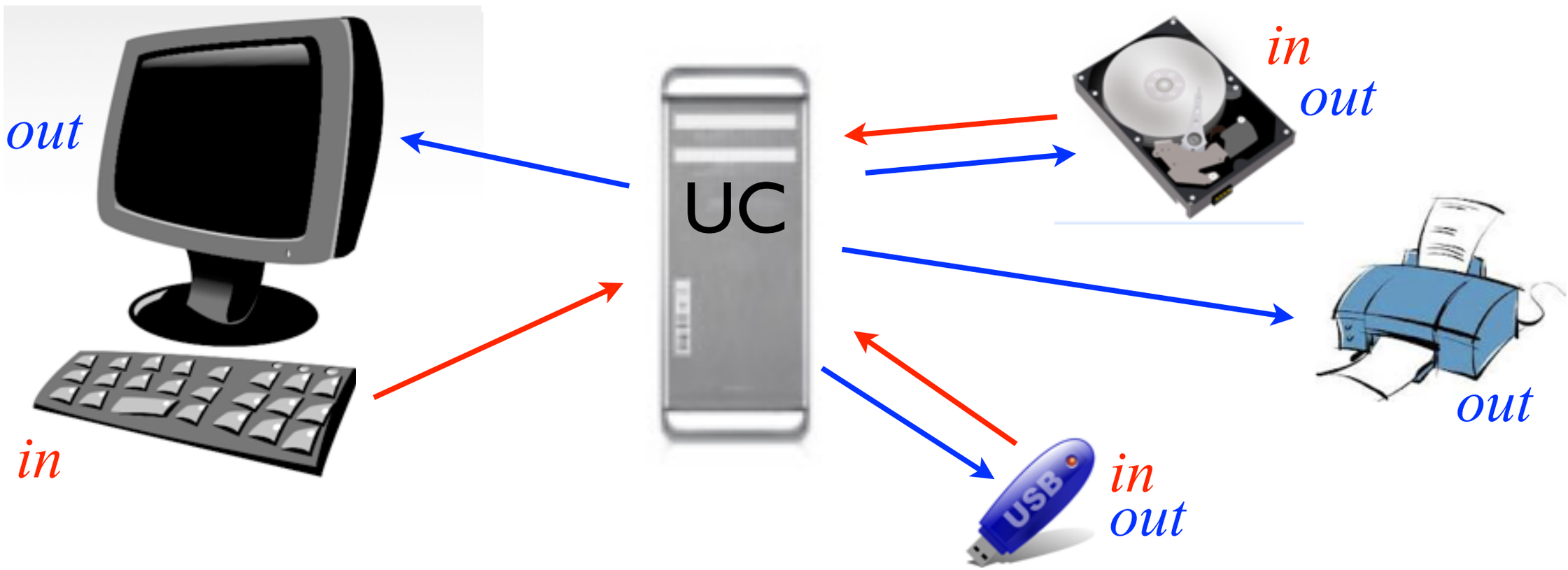
Des fichiers en entrée et en sortie

• On peut vouloir **écrire** des données dans un fichier sur le disque.

• Pour ensuite **lire** ce fichier afin d'en extraire des informations, tout ou partie des données.

• Les deux activités de base sur les fichiers sont donc :

- LECTURE (*in* : fichier en entrée)
- ECRITURE (*out* : fichier en sortie)



Ouverture d'un fichier en **écriture**

• Je 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 ne pourrons donc déposer **que des chaînes de caractères** ! Pour déposer `-234`, nous déposerons `'-234'`.

• Commençons par créer un **nouveau fichier** `test.txt` **en écriture** (*write*) avec la fonction open(*filename*,`'w'`,*encoding*) où *filename* est une chaîne contenant le chemin (absolu ou relatif) menant au fichier. Si un ancien fichier de ce nom existe déjà, il est remplacé.

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

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

• La valeur de f_out, résultat de open, est un **descripteur de fichier**. L'encodage par défaut est US-ASCII.

Ecriture dans un fichier

- Une fois le fichier ouvert, il est prêt à recevoir des données.
- On dépose un texte dans le fichier en appelant la méthode write du descripteur de fichier

```
f_out.write('hello, world!\n')
f_out.write('salut le monde!\n')
```

- 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. A la fin du traitement, il faut fermer le fichier (*close*) pour que toutes les écritures soient achevées!

```
f_out.close()
```

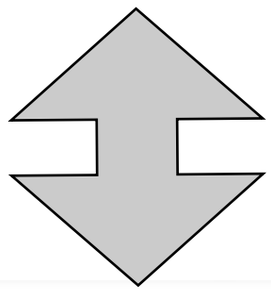
Formater les écritures

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

```
>>> f_out = open('test.txt', 'w', encoding='utf-8')
>>> f_out.write(1991)
TypeError: write() argument must be str, not int
```

- Pour déposer un nombre, il faut donc d'abord le convertir en str

```
f_out.write(str(1991))
f_out.write(' est l\'année de naissance de Python\n')
```



```
f_out.write(str(1991) + ' est l\'année de naissance de Python\n')
```

Exemple : générer une table de multiplication

```
def cree_table(n) :
    """crée un fichier contenant la table de multiplication de n """

    # etape 1: ouverture du fichier
    f_out = open("table" + str(n) + ".txt", "w", encoding="utf-8")

    # etape 2: écriture dans le fichier
    for i in range(1, 11) :
        f_out.write(str(i) + "*" + str(n) + "=" + str(i*n) + "\n")

    # etape 3 : fermeture du fichier
    f_out.close()
```

```
>>> cree_table(7)
>>>
```

```
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
```

Pour lire le fichier `table7.txt` on peut utiliser n'importe quel lecteur de fichier texte (par exemple IDLE)

Chaînes de **formatage**

Au lieu d'écrire

```
'5 * ' + str(i) + ' = ' + str(5 * i)
```

ce qui est peu lisible, on peut écrire

```
'5 * {} = {}'.format(i , 5 * i)
```

La méthode format permet de remplacer les trous {} de la **chaine de formatage** par des valeurs.

On peut aussi numéroter les trous

```
'5 * {1} = {0}'.format(5 * i , i)
```

ou encore les nommer

```
'5 * {b} = {a}'.format(a = 5 * i , b = i)
```

Formatage avancé

On peut imposer une taille au texte et un style d'alignement.

```
>>> '{0:<30}'.format('aligné gauche')
'aligné gauche                 '
>>> '{0:>30}'.format('aligné droit')
'                   aligné droit'
>>> '{0:^30}'.format('centré')
'           centré             '
>>> '{0:*^30}'.format('centré')  # remplit avec '*'
'************centré*************'
```

On peut choisir entre différents formats pour afficher un flottant

```
>>>  '{pi:.2f}; {a:.3e}; {b:.1%}'.format(pi=pi, a=7**7,
b=0.1234)
'3.14; 8.235e+05; 12.3%'
```

Table de multiplication alignée

```
def cree_table2(n) :
    """crée un fichier contenant la table de multiplication de n """

    # etape 1: ouverture du fichier
    f_out = open("table" + str(n) + « .txt", "w", encoding="utf-8")

    # etape 2: écriture dans le fichier
    for i in range(1, 11) :
        f_out.write("{:>2} * {} = {:>2}\n".format(i , n, i*n))

    # etape 3 : fermeture du fichier
    f_out.close()
```

```
>>> cree_table2(7)
>>>
```

```
 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
```

Ecrire à la fin d'un fichier

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

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

- Je vais rajouter deux lignes à la fin de mon fichier :

```
def augmenter_fichier(f) :
    f_out = open(f,'a',encoding='utf-8')
    for i in range(11,13) :
        f_out.write('{} * 7 = {}\n'.format(i, 7*i))
    f_out.close()
```

NB : Il n'est pas possible de supprimer des lignes dans un fichier. Il faut créer un nouveau fichier et détruire l'ancien !

Ouverture d'un fichier en lecture

- Problème inverse : comment lire le fichier texte `test.txt` ?
- Je dois connaître son encodage ! Or je sais qu'il est en UTF-8.

```
f_in = open('table5.txt','r',encoding = 'utf-8')
```

- Je peux lire d'un seul coup la **totalité du fichier** dans une seule chaîne de caractères, avec la méthode `read()`.

```
>>> texte = f_in.read()                    # une seule lecture !
>>> f_in.close()
>>> texte
'5 * 1 = 5\n5 * 2 = 10\n5 * 3 = 15\n5 * 4 = 20\n'
>>> print(texte)
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
```

*N.B. La **fin de ligne** est codée différemment suivant les systèmes. Sur MacOS-X et Linux, c'est '\n'. Sur Windows, c'est '\r\n'. Python s'adapte au système utilisé.*

• Ou bien je lis le fichier **ligne à ligne** avec la méthode `readline()`, jusqu'à obtention d'une ligne vide.

```
def afficher_fichier(f) :                    # ligne à ligne
    f_in = open(f,'r',encoding='utf-8')
    i = 1
    while True :
        ligne = f_in.readline()
        if ligne == '' : break    # fin du fichier !
        print(i,':\t', ligne, sep='', end='')
        i = i + 1
    f_in.close()
```

```
>>> afficher_fichier('table5.txt')
1:      5 * 1 = 5
2:      5 * 2 = 10
3:      5 * 3 = 15
4:      5 * 4 = 20
```

• Il est parfois possible d'éviter `readline()`, car un descripteur de fichier est un objet itérable :

```
def nb_lignes(f) :
    f_in = open(f,'r',encoding='utf-8')
    cpt = 0
    for ligne in f_in :
        cpt = cpt + 1
    f_in.close()
    return cpt
```

```
>>> nb_lignes('test.txt')
6
```

```
def nb_lignes(f) :
    f_in = open(f,'r',encoding='utf-8')
    cpt = sum(1 for ligne in f_in)
    f_in.close()
    return cpt
```

• Enfin, la méthode `readlines()` permet d'obtenir en une seule instruction la **liste de toutes les lignes** d'un fichier texte.

```
cpt = len(f_in.readlines())
```

Travail en lecture et écriture

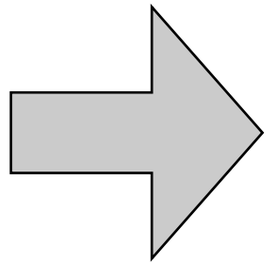
• Souvent, on prend un fichier en entrée et on produit un autre fichier en sortie (transformation).

• Exemple. J'ai un fichier dont chaque ligne contient des nombres. Je veux remplacer chaque ligne par la somme de ces nombres.

fichier en entrée

```
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
```

alea.txt



en sortie

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

somme.txt

Génération de alea.txt

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

```
def creer_alea(f) :
    f_out = open(f,'w',encoding='utf-8')
    for li in range(8) :        # je produis 8 lignes
        for co in range(5) :    # chaque ligne avec 5 colonnes
            f_out.write('{} '.format(randint(10,100)))
        f_out.write('\n')       # je vais à la ligne
    f_out.close()
```

```
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
```

Comment découper une ligne de texte ?

• Très souvent, l'information stockée dans un fichier contient divers éléments sur chaque ligne, séparés par une virgule (ou un espace, etc). C'est ce que fait Excel lorsqu'il sauve une feuille de calcul au format texte, pour que le programmeur puisse l'exploiter en programmant.

• Pour découper une ligne et obtenir une liste de ses constituants sous forme de chaînes, on utilisera la méthode `str.split(sep=' ')` :

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

• Inversement, la méthode `sep.join(L)` permet de recoller les éléments d'une liste de chaînes, avec un séparateur :

```
>>> '-'.join(['anglais','12','8','15'])
'anglais-12-8-15'
```

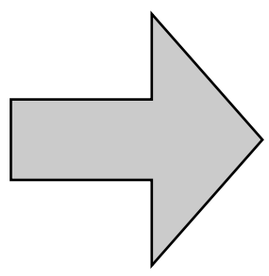
Génération de somme.txt

Pour générer le fichier somme.txt, je vais lire alea.txt et après chaque ligne lue écrire la somme de la ligne dans somme.txt.

```
f_out = open('somme.txt', 'w' ,encoding= 'utf-8')
f_in = open('alea.txt', 'r' ,encoding= 'utf-8')
for ligne in f_in :   # j'itère sur les lignes de alea.txt
  L = ligne.split()   # je récupère les entiers sur la ligne
  L2 = [int(x) for x in L]  # je les convertis en entiers Python
  f_out.write('{}\n'.format(sum(L2))  # j'écris leur somme dans
somme.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
```

alea.txt



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

somme.txt

Les exceptions liées aux fichiers

```
>>> f_in = open('existe_pas.txt', 'r', encoding='utf-8')
FileNotFoundError
>>> f_in = open('pas_droit_lecture.txt', 'r', encoding='utf-8')
PermissionError
>>> f_in = open('toto.txt', 'r', encoding='utf-8')
>>> f_in.close()
>>> f_in.read()
ValueError: I/O operation on closed file.
>>> f_out = open('pas_droit_ecriture_creation.txt', 'w', encoding='utf-8')
PermissionError
etc…
```

Il arrive souvent qu'une exception court-circuite la fermeture d'un fichier

```
f_out = open('toto.txt', 'w', encoding='utf-8')
…  2/0 …
f_out.close() # court-circuité: données perdues!
```

Pour éviter cela: Python 3 privilégie d'utiliser un bloc **with… as** qui garantit la fermeture du fichier

```
with open('toto.txt','w',encoding='utf-8') as f_out :
    … 2 / 0 …
# f_out.close() implicite en sortie du bloc with…as
# exécuté même si une exception a eu lieu
```

Résumé sur les fichiers-texte

- **Ouverture en lecture :**

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

- **Ouverture en écriture :**

```
f_out = open('foo.txt','w',encoding='utf-8')
f_out = open('foo.txt','a',encoding='utf-8')
```

- **Fermeture :**

```
f.close()    ou mieux, utiliser un bloc with…as
```

- **Lecture :**

```
f_in.read()          f_in.readline()          f_in.readlines()

for ligne in f_in : …
```

- **Ecriture :**

```
f_out.write(x)
```