

Séance 8

Mercredi 25 novembre

Les fichiers

les fichiers en python

Pourquoi des fichiers?

Un fichier va nous permettre par exemple de stocker des données, puis de pouvoir les relire et donc aussi de pouvoir les partager.

Pour utiliser un fichier, il faudra toujours

- l'ouvrir (grâce à son nom de stockage)
 - soit pour lire des données
 - soit pour écrire des données(mais pas les deux en même temps)
- et à la fin de l'utilisation le fermer.

écriture dans un fichier

Pour pouvoir écrire dans un fichier, il faut l'ouvrir en écriture et donner un nom à ce port d'écriture:

```
>>> monport= open("fichier_test","a")
```

le premier paramètre de open est le nom du fichier avec son chemin (fichier qui existe ou non, s'il n'existe pas il sera créé avec ce nom) et le deuxième correspond à la manière de l'ouvrir.

“a” pour ouvrir le fichier en mode ajout (ou append) c'est à dire que les données sont ajoutées à la fin du fichier.

“w” pour ouvrir le fichier en mode write ce qui veut dire qu'un nouveau fichier est ouvert à chaque fois (s'il existe un fichier du même nom, il est écrasé), les données sont mises au début de ce nouveau fichier vide.

On peut alors écrire dans ce “port” (= fichier ouvert en écriture) avec la méthode write:

```
>>> monport.write("bonjour tout le monde")  
>>> monport.write("au revoir à tous")
```

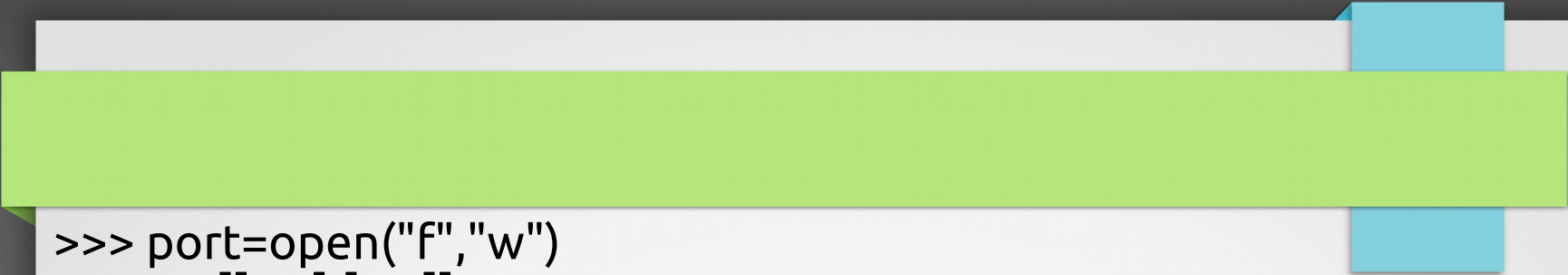
Attention on écrit dans monport pas dans fichiertest

Les données sont écrites les unes à la suite des autres sans espace (sauf si on en rajoute)

Attention on ne peut écrire que des strings

A la fin, il faut ***impérativement*** fermer le fichier grâce à la méthode close (sans argument). Il sera ensuite utilisable pour tout usage.

```
>>> monport.close()
```

```
>>> port=open("f","w")
>>> m=[[1,2],[2,5]]
>>> port.write(m)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    port.write(m)
TypeError: must be str, not list
```

```
>>> d={3:"truc"}
>>> port.write(d)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    port.write(d)
TypeError: must be str, not dict
```

```
>>> port.write(5)
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    port.write(5)
TypeError: must be str, not int
```

Nous allons créer un fichier pour les tests:

```
f=open("fichier_nombre","w")  
for i in range (1,10):  
    f.write(str(i)+ " ")  
f.write ("fin")  
f.close()
```

On aura alors dans le fichier exactement:

1 2 3 4 5 6 7 8 9 fin

lecture dans un fichier

On peut ouvrir un fichier en lecture pour y lire des données. C'est toujours la fonction `open` mais avec "r" (comme read) comme deuxième paramètre.

```
>>> l=open("fichier_nombre","r")
>>> truc=l.read()
>>> truc
"1 2 3 4 5 6 7 8 9 fin"
```

Comme on n'a pas mis d'argument à `read`, le fichier est lu dans son intégralité et la string correspondante est renvoyée.

Attention: "it's your problem if the file is twice as large as your machine's memory"



Comme pour l'écriture, il faut impérativement fermer le fichier après utilisation. C'est encore la méthode `close` qui s'applique.

```
>>> l.close()
```

On peut aussi mettre un argument à `read`: le nombre de caractères à lire, ce nombre de caractères est alors compté à partir de la position actuelle dans le fichier.

Si on reprend le fichier nommé fichier_nombre

```
>>> lect= open("fichier_nombre","r")
>>> t1=lect.read(8)
>>> t1
'1 2 3 4 '
>>> t2=lect.read(6)
>>> t2
'5 6 7 '
>>> t3=lect.read(14)
>>> t3
'8 9 fin'
>>> t4=lect.read(1)
>>> t4
''
```

On voit bien que le deuxième read commence la lecture là où s'est arrêté le premier read. S'il ne reste pas assez de caractères alors le read s'arrête à la fin du fichier (sans erreur).

Si on était déjà à la fin du fichier c'est une chaîne vide qui est renvoyée.



Attention: il vaut mieux prendre l'habitude de bien fermer les fichiers à chaque fois

- le nombre de fichiers simultanément ouverts est limité
- Python ne garantit pas que les données écrites le soient vraiment tant que le fichier n'a pas été fermé proprement...
- N'oubliez pas de fermer un fichier après l'avoir ouvert. Si d'autres applications, ou d'autres morceaux de votre propre code tentent d'ouvrir un fichier qui n'a pas été fermé, ils ne pourront pas car le fichier sera déjà ouvert.

Exemple de programmes avec des fichiers:

On veut écrire un petit programme qui compare deux fichiers et renvoie True si les fichiers sont identiques ou sinon les premiers caractères différents rencontrés.

```
def compare(fic1,fic2):  
    f1= open(fic1,"r")  
    f2=open(fic2,"r")  
    while 1:  
        x1,x2=f1.read(1),f2.read(1) # lecture 1 caractère de chaque fichier  
        if x1!= x2:  
            f1.close()  
            f2.close()  
            return x1,x2 # d'abord fermer avant de faire return  
        if x1=="": #si on arrive là fin de fichier pour les 2  
            f1.close()  
            f2.close()  
            return True
```

remarque: avec while 1 on lance une boucle infinie qui sera arrêtée par un return, soit en fin de fichier soit quand on rencontre deux caractères différents

remarque: sortie de boucle avec break

On a écrit une boucle while un peu spéciale puisque la condition du while est ici 1: c'est une condition toujours "vraie" qui n'évolue pas; donc on lance alors une boucle qui est infinie à priori!!!

Attention, syntaxiquement il faut toujours une condition dans un while

Dans le cas du programme précédent, il se terminait parce qu'on avait toujours un return de prévu.

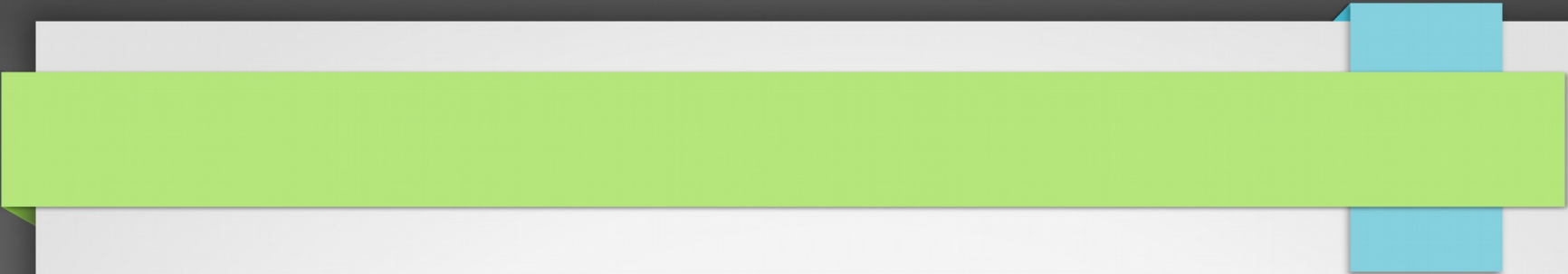
Quand on parcourt des fichiers on aura souvent ce genre de condition toujours vérifiée, avec un arrêt prévu soit "'à la fin du fichier" soit jusqu'à rencontrer un caractère soit ...

Si le programme ne nécessite pas le renvoi d'une donnée, on peut interrompre la boucle avec l'instruction break (sans paramètre et sans parenthèses).

exemple: recopier un fichier dans un autre fichier, caractère par caractère:

```
def recopie(source,but):  
    f1= open(source,"r") #ouvert en lecture  
    f2=open(but,"w") # ouvert en écriture  
    while 1:  
        x1=f1.read(1)  
        if x1=="": #test la fin du fichier  
            f1.close()  
            f2.close()  
            break #il n'y a rien à retourner  
        else:  
            f2.write(x1)
```

remarque : ici on a `x1==""` comme test de fin (qui ne peut pas se produire avant la fin) à ne pas confondre avec `x1==" "` (qui peut se produire dans le fichier)



remarque: la fonction break n'est pas liée à l'utilisation de fichiers, cela peut arrêter une boucle dans le cadre d'autres programmes (ne pas en abuser cependant)

En utilisant with

Si on oublie de fermer un fichier à la fin d'un programme ou si le programme plante avant de fermer un fichier on risque des problèmes ultérieurs: on peut éviter ces problèmes avec with:

```
with open('fichier.txt', 'r') as mon_fichier:  
    texte = mon_fichier.read()
```

Le fichier a été lu, puis le fichier a été fermé.

On peut le vérifier avec l'instruction `truc.closed` qui renverra `True` ou `False`

Sauvegarder et récupérer des nombres

problème: on a une liste de nombres entiers que l'on veut pouvoir sauvegarder puis retrouver ultérieurement. Si on écrit:

```
def sauv(l,fichier_sauvegarde):  
    but=open(fichier_sauvegarde,"w") # nouveau fichier  
    but.write(str(l)) # Il faut une string  
    but.close()
```

Puis

```
def recup (fichier_sauvegarde):  
    source=open(fichier_sauvegarde,"r")  
    x=source.read()  
    source.close()  
    return x
```

```
>>> sauv([1,2,3,4,5,6],"s")  
>>> recup("s")  
'[1, 2, 3, 4, 5, 6]'
```

On ne récupère pas une liste mais une string qu'il faudrait retraiter pour récupérer vraiment la liste..... pas top!

Mais possible.....

```
def traite(truc):  
    m=truc[1:-1]  
    x=m.split(',')  
    return[int(y) for y in x]
```

Souvent on écrit plutôt élément par élément:

```
def sauve(l,fichier_sauvegarde):  
    but=open(fichier_sauvegarde,"w") # nouveau fichier  
    for x in l:  
        but.write(str(x)+" ")  
    but.close()
```

Pour récupérer ces nombres, on peut

→ récupérer tout le fichier en une seule chaîne de caractères et ensuite exploser cette chaîne avec la méthode split pour récupérer les sous-chaînes correspondant à des nombres que l'on re-transforme en entiers

```
def recupere (fichier_sauvegarde):  
    source=open(fichier_sauvegarde,"r")  
    x=source.read()  
    source.close()  
    return[int(t) for t in x.split()]
```

Remarque: on a alors `x="1 2 3 4 5 6 "`
et donc si `l=x.split()` alors `l=["1","2","3","4","5","6"]`
`recupere` renverra donc bien `[1,2,3,4,5,6]`

Ou alors

→ faire une lecture caractère par caractère pour reconstituer ces nombres (l'espace indiquant la fin d'un nombre)

```
def recupere2 (fic):  
    source=open(fic,"r")  
    res, encours=[ ], ""  
    while 1:  
        x=source.read(1)  
        if x=="":  
            source.close()  
            return res  
        elif x==" "  
            res=res+[int(encours)]  
            encours=""  
        else:  
            encours=encours+x
```

remarque: ici par construction, le fichier finit forcément par un nombre suivi d'un espace (d'où le programme).

Remarque:

Nous verrons plus loin une méthode qui peut s'avérer plus simple pour sauvegarder et récupérer une liste quelconque (voire une grille ou un dictionnaire) en utilisant le module **pickle**.

Attention: Le programme précédent est uniquement adapté à une liste d'entiers pas une liste quelconque.

Les fichiers "textes"

On peut organiser certaines données dans un fichier comme par exemple pour le bottin ci-dessous, par lignes que l'on séparera par des caractères spéciaux non imprimables appelés "fin de ligne" qui sont `\n`.

```
dupont 02 31 78 67 67
durand 06 67 45 34 23
marie 03 34 23 23 23
martin 04 90 89 54 34
petit 08 78 89 79 79
```

```
>>> l=open("monbottin","r")
>>> truc=l.read()
>>> truc
'dupont 02 31 78 67 67\ndurand 06 67 45 34 23\nmarie 03 34 23 23 23\nmartin 04 90 89 54 34\npetit 08 78 89 79 79\n'
>>> l.close()
```

On peut rajouter des données:

```
>>> b=open("bottin","a")
>>> b.write("zorro 06 78 45 34 34 \n")
>>> b.close()
```

Attention à ne pas oublier le \n (saut de ligne)

On supposera toujours que les fichiers sont bien écrits

On obtient alors dans le fichier:
(en l'ouvrant avec un éditeur de texte quelconque...)

```
dupont 02 31 78 67 67
durand 06 67 45 34 23
marie 03 34 23 23 23
martin 04 90 89 54 34
petit 08 78 89 79 79
zorro 06 78 45 34 34
```

Comment lire un tel fichier?

Toutes les lignes n'ont pas obligatoirement le même nombre de caractères. On pourrait bien sûr utiliser `read` et reconstituer les lignes en cherchant le caractère `\n`.

Ce programme peut s'écrire mais en fait il existe une fonction permettant la lecture ligne par ligne: la méthode `readline()` lit une ligne (incluant le caractère de fin de ligne).

```
>>> l=open("bottin","r")
>>> l.readline()
'dupont 02 31 78 67 67\n'
>>> l.readline()
'durand 06 67 45 34 23\n'
```

Chaque appel de `readline` va aller lire la ligne suivante.....

Exemple:

on veut compter les personnes présentes dans le bottin:
on va compter les lignes en fait

```
def combien():  
    l=open("bottin","r")  
    c=0  
    while 1:  
        x=l.readline()  
        if x=="":  
            l.close()  
            return c  
        else:  
            c=c+1
```


Et si on veut afficher les noms au fur et à mesure:

```
>>> lesnoms()  
dupont  
durand  
marie  
martin  
petit  
zorro
```

remarque: pour la méthode split pour les strings:

```
>>> ch="dupont 03 45 56 23 23 \n"  
>>> ch.split()  
['dupont', '03', '45', '56', '23', '23']
```

le split va “négliger” le caractère \n

D'où le programme:

```
def lesnoms():  
    l=open("bottin","r")  
    while 1:  
        x=l.readline()  
        if x=="":  
            l.close()  
            break # pour arrêter le programme  
        else:  
            truc=x.split()  
            print truc[0]  
            # x vaut "dupont 02 31 78 67 67"  
            # truc vaut ["dupont","02","31","78","67","67"]  
            # truc[0] vaut "dupont"
```

Tout est fait au fur et à mesure, ligne par ligne

La méthode readlines()

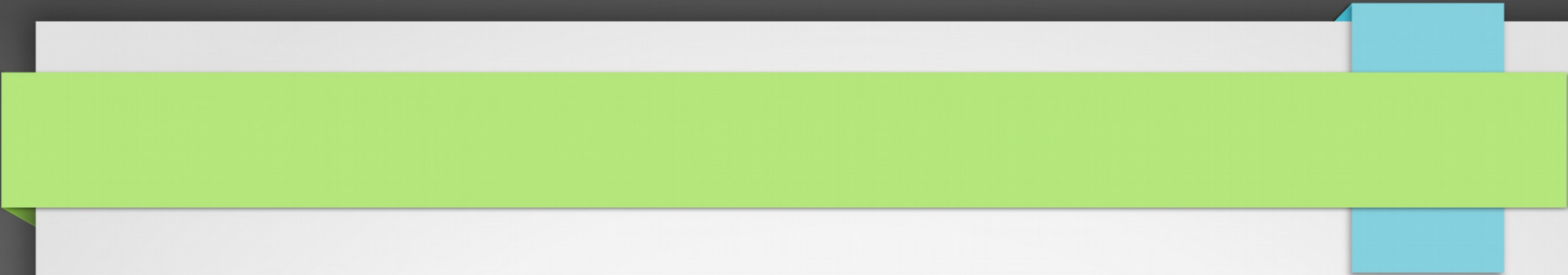
permet de récupérer les lignes “restantes” (ou toutes les lignes si c'est la première instruction): on obtient alors une liste de chaînes de caractères, chaque chaîne étant une ligne.

```
>>> l=open("bottin","r")
>>> l.readline()
'dupont 02 31 78 67 67\n'
>>> l.readline()
'durand 06 67 45 34 23\n'
>>> fin=l.readlines()
>>> fin
['marie 03 34 23 23 23\n', 'martin 04 90 89 54 34\n', 'petit 08 78 89 79 79\n', '\n', 'zorro 06 78 45 34 34 \n']
>>> l.close()
```

Cette méthode **readlines** permet donc de récupérer l'intégralité d'un texte en une seule fois: mais attention ce n'est possible que si le texte n'est pas trop important par rapport à la taille de la mémoire vive de l'ordinateur: il vaut mieux parfois faire une boucle de `readline()`

Attention: `readline` renvoie une chaîne de caractères alors que `readlines` renvoie une liste de chaînes de caractères.

```
def combien2():  
    l=open("bottin","r")  
    x=l.readlines()  
    l.close()  
    return len(x)
```

```
def lesnoms2():  
    l=open("bottin","r")  
    x=l.readlines()  
    l.close()  
    for n in x:  
        truc=n.split()  
        print truc[0]
```

On ouvre le fichier le temps de récupérer la liste des lignes, on peut le refermer aussitôt et travailler sur cette liste.

Le seul problème est de potentiellement générer une trop grosse liste!!!

for et les fichiers:

On retrouve notre "for" magique!!!
Pour les fichiers textes ce sera très efficace:

for x in fic:

(si fic est un port ouvert en lecture) reviendra à faire une boucle de readline sans l'écrire ni avoir besoin d'écrire de test d'arrêt
En plus le programme est informatiquement plus efficace!!!

```
def combien3(fichier):  
    l=open(fichier,"r")  
    c=0  
    for x in l:  
        c=c+1  
    l.close()  
    return c
```

Attention à ne pas oublier de fermer le fichier

il faut le faire avant l'instruction return !!

```
def lesnoms3(fichier):  
    l=open(fichier,"r")  
    for x in l:  
        truc=x.split()  
        print truc[0]  
    l.close()
```

Plus besoin de boucle "infinie" et de test de fin....

Attention ce n'est valable **que** pour les fichiers **textes**
car on travaille par lignes...

Ou alors

```
def lesnoms3(fichier):  
    with open('fichier.txt', 'r') as l:  
        for x in l:  
            truc=x.split()  
            print truc[0]
```


for en compréhension

On peut utiliser le for en compréhension:
Si on veut la liste des noms du bottindu fichier fic:

```
def lesnoms4(fic):  
    l=open(fic,"r")  
    truc=[x.split()[0] for x in l]  
    l.close()  
    return truc
```

Attention à la place du return et du close

Encore quelques exemples sur le bottin

On veut recopier dans un nouveau fichier les numéros un par un en demandant à l'utilisateur au fur et à mesure s'il veut garder le numéro

```
>>> recopieAvecTest("monbotin","newmonbottin")
```

```
dupont 00 33 2 31 78 67 67
```

```
tapez 1 pour le garder 0 sinon 1
```

```
durand 00 33 6 67 45 34 23
```

```
tapez 1 pour le garder 0 sinon 1
```

```
marie 00 33 3 34 23 23 23
```

```
tapez 1 pour le garder 0 sinon 0
```

```
martin 00 33 4 90 89 54 34
```

```
etc....
```

→ Première possibilité:
on écrit dans un "nouveau fichier"

```
def recopietest(bot,newbot):  
    b1=open(bot,"r")  
    b2=open(newbot,"w")  
    for x in b1: # recuperation ligne par ligne  
        print x  
        r=input("tapez 1 pour le garder 0 sinon")  
        if r=="1":  
            b2.write (x)  
    b1.close()  
    b2.close()
```

Inconvénient: on a crée un nouveau fichier alors qu'on aurait préféré garder l'ancien

→ avec l'utilisation de readlines

on peut commencer par récupérer toutes les lignes puis ensuite tamponner le fichier et réécrire les nouvelles données dedans.

```
def recopietest2(bot):  
    b=open(bot,"r")  
    liste= b.readlines()  
    b.close()  
    b=open(bot,"w")  
    for x in liste: # ici le for un parcours de liste  
        print x  
        r=input("tapez 1 (garder)ou 0 (sinon)")  
        if r=="1":  
            b.write (x+ "\n")  
    b.close()
```

Ici on peut facilement garder le même fichier . Ce qui n'est pas possible dans la version précédente (car dans l'autre version on lit et on écrit en même temps)

Fichiers et dictionnaires

Pour pouvoir sauvegarder un dictionnaire, il faudrait pouvoir l'écrire dans un fichier..... la plupart du temps on va en faire un fichier texte avec une donnée clé valeur sur chaque ligne.

on considère par exemple le dictionnaire:

```
score={'toto': 30, 'titi': 23, 'riri': 10, 'fifi': 14, 'loulou': 40}
```

On veut l'écrire dans un fichier texte sous la forme suivante:

```
toto 30  
titi 23  
riri 10  
fifi 14  
loulou 40
```

D'où le programme pour la sauvegarde:

```
def ecrire(dico,fichier_sauve):  
    p=open(fichier_sauve,"w")  
    for x in dico:  
        p.write (x + " " + str(dico[x]) + "\n")  
    p.close()
```

Si un fichier texte est écrit de cette manière:

il faut alors écrire une fonction pour “récupérer” le dictionnaire c'est à dire le reconstruire.

```
def lire(fichier_sauve):  
    dic={}  
    p=open(fichier_sauve,"r")  
    for x in p:  
        ligne=x.split()  
        cle,val=ligne[0], int(ligne[1])  
        dic[cle]=val  
    p.close()  
    return dic
```

Remarque:

comme pour les listes on peut aussi utiliser le module pickle (voir plus loin) pour sauvegarder et récupérer un dictionnaire.

Résumé

→ fichiers contenant des données “connues”, un texte ..
read(1): traitement caractère par caractère
read(): traitement global
read(p): si on connaît la taille des données

Dans tous les cas on écrit avec write

→ fichiers textes

readlines()

readline()

boucle for

writelines

remarque: writelines(liste_de_chaines) peut écrire toutes les lignes en une seule fois (mais il faut que les lignes contiennent le \n à la fin)

Le module pickle

Le module pickle permet d'écrire puis de relire des données "compliquées" dans un fichier à l'aide des méthodes dump pour écrire et load pour relire.

Exemple:

```
import pickle

def sauv(l,f):
    p=open(f, "wb") # sauvegarde dans f
    pickle.dump(l,p)
    p.close()

def lit(f):
    p=open(f,"rb")
    truc= pickle.load(p)
    p.close()
    return truc
```

```
>>> sauv(listemot)
```

```
>>> lit("f")
```

```
['ABRICOT', 'POMME', 'POIRE', 'ORANGE', 'CERISE', 'FRAISE',  
'FRAMBOISE', 'KIWI', 'KUMQUAT', 'MYRTILLE', 'ANANAS']
```

```
>>> sauv(grilmot)
```

```
>>> lit("f")
```

```
[['Y', 'O', 'O', 'N', 'S', 'J', 'G', 'D'], ['U', 'A', 'J', 'V', 'E', 'Q', 'Y', 'E'], ['C',  
'B', 'A', 'K', 'T', 'P', 'W', 'T'], ['M', 'Y', 'T', 'N', 'Y', 'E', 'E', 'F'], ['C', 'Y', 'J',  
'W', 'R', 'B', 'P', 'I'], ['K', 'F', 'J', 'E', 'T', 'D', 'B', 'P'], ['N', 'O', 'S', 'X', 'F',  
'T', 'S', 'T'], ['F', 'X', 'Y', 'A', 'H', 'H', 'N', 'S']]
```

```
>>> sauv(score)
```

```
>>> lit("f")
```

```
{'toto': 30, 'titi': 23, 'riri': 10, 'fifi': 14, 'loulou': 40}
```

Remarques:

- Le fichier doit absolument être relu avec load (codage non lisible)
- Le module pickle est évidemment intéressant pour une sauvegarde /relecture de données dans un programme, mais cela ne remplace pas les fichiers textes....
- l'avantage de stocker les scores dans un fichier texte plutôt que de sauvegarder le dictionnaire avec pickle est une lecture possible du fichier texte par autre chose qu'un programme python

Exemple d'utilisation du mécanisme d'exceptions pour des fichiers

On veut ouvrir un fichier en lecture, pour récupérer son contenu. Le nom du fichier est demandé à l'utilisateur:

Sans gestion des exceptions, le programme est:

```
def essai1():  
    fichier = input(" nom du fichier? : ")  
    f = open(fichier, "r")  
    lignes=f.readlines()  
    f.close()  
    return lignes
```

Et le programme fonctionne très bien dès lors que le fichier existe....mais sinon.... on voudrait que le programme ne se plante pas même si le fichier n'existe pas.

IOError

Remarque: pour l'instant, si le fichier saisi par l'utilisateur n'existe pas, on sort du programme, et on obtient le message d'erreur:

```
>>> essai1()
```

Veillez entrer un nom de fichier : f

Traceback (most recent call last):

File "<pyshell#14>", line 1, in -toplevel-
essai1()

File "/home/fse/mon python/cours10B.py", line 73, in essai1
f = open(fichier, "r")

IOError: [Errno 2] No such file or directory: 'f'

On va donc “essayer” d'ouvrir le fichier en lecture en “interceptant” une éventuelle erreur liée à cet essai puis gérer l'exception, ce qui donne:

```
def essai():  
    fichier = input("entrer un nom de fichier: ")  
    try:  
        f = open(fichier, "r")  
  
    except IOError:  
        print "Le fichier", fichier, " est introuvable"  
  
    else:  
        lignes=f.readlines()  
        f.close()  
        return lignes
```

permettre à l'utilisateur de recommencer:

```
def essai2():  
    fichier = input(" nom du fichier? : ")  
    try:  
        f = open(fichier, "r")  
    except IOError:  
        print "Le fichier", fichier, "est introuvable"  
        return essai2()  
    else:  
        lignes=f.readlines()  
        f.close()  
        return lignes
```

```
>>> essai2()  
Veuillez entrer un nom de fichier : jkl  
Le fichier jkl est introuvable  
Veuillez entrer un nom de fichier : hjk  
Le fichier hjk est introuvable  
Veuillez entrer un nom de fichier : notes  
['dupont 12 14 15 12\n', 'durand 5 8 3 9\n', 'marie 14 17 18 19\n', 'martin 12 15  
11 19\n']
```