

# **Cours n°6**

**mercredi 4 novembre**

**Travailler avec des grilles**

# Travailler avec des grilles

On utilise des “grilles” pour représenter un certain nombre de données:

- des tableaux de nombres  
qui peuvent être des matrices, des images
- des grilles de nombreux jeux (qu'elles apparaissent ou non à l'écran)

# une grille

On représentera une grille par une liste de listes, généralement une liste de lignes

On supposera toujours que toutes les sous-listes ont la même taille

exemple

```
[[0,0,1,2],[2,1,0,0],[0,0,0,0]]
```

# Fonctions de base à écrire

# pour trouver ce qu'il y a à la place i,j de grille

```
def trouve(i,j,grille):
```

```
    return grille[i][j] # m=grille[i] est une liste et on prend m[j]
```

#pour modifier ce qu'il y a à la place i,j

```
def change(i,j,grille,x):
```

```
    grille[i][j]=x # rien n'est renvoyé,
```

```
    # grille est modifiée
```

# création de grilles

On veut écrire une fonction capable de créer une grille à n lignes et p colonnes remplie de 0. Si la fonction s'appelle cree\_grille(n,p) elle doit créer et renvoyer la grille suivante

```
>>>cree_grille(3,4)
[[0,0,0,0],[0,0,0,0],[0,0,0,0]]
```

Première idée : (fausse)

```
def gril(n,p):
    l= ([0]*p)
    return [l]*n
```



# création de grilles

Le problème est que chaque sous-liste est la même et non pas une copie de même valeur.

```
>>> m=grille(6,3)
>>> m
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> change(1,1,m,8)
>>> m
[[0, 8, 0], [0, 8, 0], [0, 8, 0], [0, 8, 0], [0, 8, 0], [0, 8, 0]]
>>> change(0,0,m,2)
```

## bonne version pour une “vraie” grille !

```
def creation_grille (n,p):  
    g=[0]*n  
    for i in range(n):  
        g[i]=[0]*p
```

Il faut mettre une liste différente dans chaque case (même si toutes ces listes ont la même valeur, ce ne doit pas être la même liste)

Ces “grilles” serviront pour de nombreux jeux mais aussi pour des images ou des matrices.

# Doubles boucles

On a besoin de doubles boucles pour travailler avec des grilles.

Exemple: on suppose que la grille est remplie de nombres et on voudrait en faire la somme

```
def somme1(grille):  
    n=len(grille)  
    p=len(grille[0])  
    res=0  
    for i in range(n):  
        for j in range(p):  
            res+=grille[i][j]  
    return res
```

```
def somme2(grille):  
    res=0  
    for ligne in grille:  
        for x in ligne:  
            res+=x  
    return res
```



Et si on veut tester qu'une grille ne comporte aucun 0:

```
def paszero2(grille):  
    for ligne in grille:  
        for x in ligne:  
            if x==0:  
                return False  
    return True
```

```
def paszero1(grille):  
    n,p=len(grille),len(grille[0])  
    res=0  
    for i in range(n):  
        for j in range(p):  
            if grille[i][j]==0:  
                return False # là on pourrait renvoyer (i,j)  
    return True
```

# Utiliser un module personnel

Imaginons qu'on ait écrit plusieurs fonctions sur des grilles dans un fichier appelé `mesGrilles.py`

```
# creation de la grille
def creegrille(n,p, x): # carrée de taille n sur p avec x dans toutes les cases
    res=[0]*n
    for i in range(n):
        res[i]= [x]*p
    return res

def nblignes(grille):
    return len(grille)

def nbcols(grille):
    return len(grille[0])

# affichage mode console
def affiche (grille):
    for l in grille:
        for x in l:
            print(x,end=" ")
        print()
```

# Utiliser un module personnel

On peut ensuite s'en servir dans d'autres programmes sans copier/coller

```
from mesGrilles import *   # attention on ne met pas le .py

from random import randint

# re-initialisation d'une matrice e, en mettant 0 ou 1 dans toutes les cases
def reinitAlea(m):
    n,p=nblignes(m), nbcols(m)
    for i in range(n):
        for j in range(p):
            m[i][j]=randint(0,1)
    return m

# creation d'une matrice n ,p avec des nombres alea entre 1 et 9 inclus
def initAlea(n,p):
    m= creegrille(n,p,0)
    for i in range(n):
        for j in range(p):
            m[i][j]=randint(1,9)
    return m
```

## Des tests

Jusqu'à présent, on a souvent supposé que les programmes seraient toujours bien utilisés et en particulier qu'il n'y aurait pas d'erreur de saisie ni de "triche" ou de mauvaise utilisation même si on a déjà mis quelques tests en place.

C'est bien sûr inimaginable et nous allons donc nous préoccuper des tests à faire.

Attention: il ne s'agit pas de pallier à des erreurs de programmation!!! mais des erreurs d'utilisation.....

Nous allons cependant revoir TOUS les cas d'erreurs



# les erreurs de syntaxe

C'est les premiers problèmes rencontrés!!!  
dès la lecture du programme

On ne peut pas continuer d'exécuter un programme en contenant au moins une!!

Ces erreurs proviennent généralement des problèmes suivants:

- mauvaise indentation, oubli de :
- mauvaise utilisation des mots clés
- parenthèses non fermées
- apostrophes ou guillemets non fermés ou absents.....



Il faut commencer par corriger ces erreurs de syntaxe!!!

Pour cela il faut lire les messages d'erreur!!!

exemple:

```
>>> while 1 print "bonjour"  
SyntaxError: invalid syntax
```

Le lieu où commence le problème est en rouge.  
Ce n'est pas print qui est faux mais à cet endroit on ne devrait pas arriver sur print car il manque le caractère: après le while....

Une fois qu'on a enlevé les erreurs de syntaxe on peut commencer à exécuter le programme.....

# les problèmes à l'exécution

Ce n'est pas parce qu'un programme est syntaxiquement correct que tout va bien!!! Il peut y avoir une erreur à l'exécution

- parce que l'on se sert mal du programme
- parce qu'il est mal écrit
- parce qu'il y a un cas non prévu
- parce que l'utilisateur fait une faute de frappe
- parce qu'une variable n'a pas de valeur
- parce qu'une fonction n'a pas été définie

.....

## exemple d'erreur typique :

(peut se trouver au coeur d'un programme)

```
>>> ch="6+1o"
```

```
>>> eval(ch)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#3>", line 1, in <module>
```

```
    eval(ch)
```

```
File "<string>", line 1
```

```
    6+1o
```

```
    ^
```

```
SyntaxError: unexpected EOF while parsing
```

## Exemple de problèmes

On avait un petit programme de calcul de somme des entiers de 1 à n qui fonctionnait très bien à condition bien entendu qu'on l'appelle avec un entier.

```
def somme(n):  
    som=0  
    for i in range(n+1):  
        som+=i  
    return som
```

On transforme ce programme pour demander à l'utilisateur la valeur de n. Ce qui donne la version sans test:

```
def test():  
    n=input("pour quelle valeur de n ? ")  
    return somme(int(n))
```

## Que peut-on faire comme erreur à la saisie?

→ si on ne tape pas un "entier", il y aura un problème avec int après le input

→ si c'est un nombre négatif ce n'est pas une bonne idée (mais cela renvoie 0)



## Quelques essais :

```
>>> test()
quel terme voulez-vous 3è
```

Traceback (most recent call last):

```
File "<pyshell#5>", line 1, in -toplevel-
    test2()
```

```
File "/home/fse/mon python/cours10B.py", line 28, in test1
    n=input("quel terme voulez-vous ")
```

```
File "<string>", line 1
    3è
    ^
```

SyntaxError: unexpected EOF while parsing

Il faudrait donc pouvoir tester que l'utilisateur a fait une bonne saisie mais surtout, que cette saisie ne se plante pas!!!

**C'est cette fonction de saisie qui doit s'occuper des tests et non pas la fonction somme.**

Ici , on utiliserait cette saisie de la façon suivante:

```
def test():  
    return somme(saisie())
```

→ Si la saisie d'un nombre a réussi.... on peut tester que c'est un entier positif...

→ pour "réussir" la saisie on peut tester que la chaîne saisie par input ne contient que des caractères chiffres (déjà fait en cours)

→ mais comment réussir si on permet de saisir une expression style 3+5.... cela se complique!!!

On va utiliser le mécanisme d'exception de Python

# les exceptions

Les exceptions sont l'essentiel des erreurs qui se produisent lors de l'exécution des programmes. Il y en a de plusieurs types (et on peut en créer). Les plus fréquentes sont:

- ArithmeticError
- ZeroDivisionError
- OverflowError
- IndexError
- TypeError
- ValueError
- EnvironmentError
- IOError

# gestion des exceptions

En utilisant l'ensemble d'instructions:

try – except - else

on va pouvoir intercepter une erreur “prévue” dans le programme, et continuer l'exécution sans erreur.

```
def saisie():
```

```
    try:
```

```
        n=int(input ("quel terme ? "))
```

```
    except ValueError:
```

```
        print (" on avait dit un entier")
```

```
        return saisie()
```

```
    else:
```

```
        if n>0:
```

```
            return n
```

```
        else:
```

```
            print ("l'entier doit être positif")
```

```
            return saisie()
```



## Que se passe-t-il?

**try:**

**n=int(input ("quel terme ?"))**

le programme "essaye" de transformer la saisie en entier

**except ValueError:**

s'il y a une exception ValueError, l'erreur ne se déclenche pas et n n'est pas calculé et ces instructions sont alors exécutées:

**print (" on avait dit un entier")**

**return saisie()**

**else:**

s'il n'y a pas eu d'erreur le programme peut continuer avec les instructions qui suivent

**if n>0:**

**return n**

**else:**

**print ("l'entier doit être positif")**

**return saisie()**



Et cela marche très bien:

```
>>> saisie()
quel terme ? (entier positif)-4
l'entier doit être positif
quel terme ? (entier positif)hjk
on avait dit un entier
quel terme ? (entier positif)7
28
```

# Syntaxe de la gestion des exceptions

**try:**  
    **instructions à essayer**  
**except <type d'erreur>:**  
    **instructions au cas où une exception est déclenchée**  
**else:**  
    **instructions si aucune exception n'a été déclenchée**

Ce mécanisme fonctionne comme suit :

Le bloc d'instructions qui suit directement une instruction **try** est exécuté par Python *sous réserve*.

Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction **except**. Si aucune erreur ne s'est produite dans les instructions qui suivent **try**, alors c'est le bloc qui suit l'instruction **else** qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

# Remarques

- On peut faire suivre l'instruction **try** de plusieurs blocs **except**, chacun d'entre eux traitant un type d'erreur spécifique. Au plus une clause d'exception sera exécutée.
- Dans une clause d'exception ne seront gérées que les exceptions survenant dans la clause d'essai correspondante, et non pas celles provenant d'autres clauses d'exception.
- Une clause d'exception peut nommer plusieurs exceptions dans une liste parenthésée, par exemple:  
... except (RuntimeError, TypeError, NameError):

## Attention

L'instruction `try...except` doit impérativement contenir un `else` pour placer le code qui doit être exécuté si la clause d'essai ne déclenche pas d'exception.



## Retour sur le jeu du nombre mystérieux :

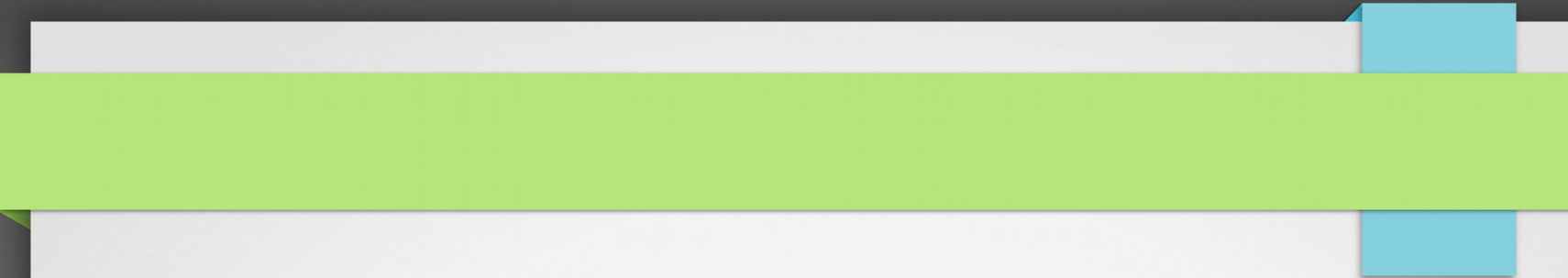
```
def jeu():
    x=randint(1,101)
    c, gagne = 0,0
    while not gagne:
        nb=int(input("entrer votre nombre ou -1 pour stopper"))
        c=c+1
        if nb== -1:
            print "paresseux, il fallait trouver", x
            break
        elif nb<x:
            print (nb , " c'est trop petit")
        elif nb == x:
            print ("bravo c'est gagne en ",c," coups")
            gagne=1
        else:
            print (nb, "c'est trop grand")
```

Dans le jeu du nombre mystérieux version texte , le seul problème qui peut se poser est lié à la saisie du nombre par l'utilisateur: il doit correspondre à un entier .....



## Si on utilise le mécanisme d'exception:

```
def jeu():  
    x=randint(0,100)  
    gagne,c = 0,0  
    while not gagne:  
        n=input("entrer un nombre,-1 pour stopper")  
        c=c+1  
        try:  
            nb=int(n)  
        except ValueError:  
            print "erreur de saisie"  
        else:  
            if nb== -1:  
                suite identique au programme précédent
```



Comme il y a une boucle while, si une exception est déclenchée nb ne sera pas modifié et donc on aura toujours `nb != x` donc la boucle continue et une nouvelle saisie est automatiquement demandée (ici, on comptera quand même un coup pour l'erreur ce qui est modifiable si on le souhaite)

Attention: si vous rajouter des mots clés pour certaines actions (par exemple `help` pour avoir une aide) il faut faire les tests avant le `try`

Cette version peut accepter de saisir un float ou une opération en remplaçant `int` par `float` ou `eval`

```
def jeu():
    x=randint(0,100)
    gagne,c = 0,0
    while not gagne:
        n=input("entrer un nombre,-1 pour stopper,help pour aide")
        if n=='help':
            Traitement à écrire
        else :
            c=c+1
            try:
                nb=int(n)
            except ValueError:
                print "erreur de saisie"
            else:
                if nb==-1:
                    suite identique au programme précédent"
```