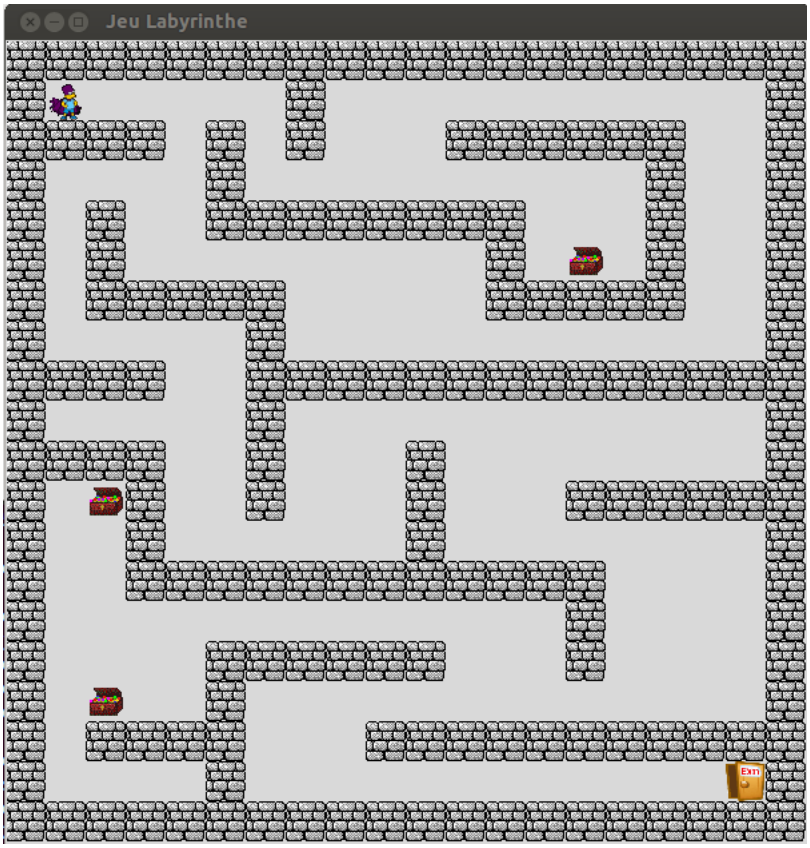
Projet Jeu de plateforme sous Python

Python en ISN

M. Lagrave

Le projet

Pour apprendre à programmer en Python, nous allons créer en sept séances un petit jeu, étape par étape, en faisant intervenir des notions de plus en plus avancées.



Notre objectif est de concevoir un petit jeu dans lequel il faudra déplacer à l'aide du clavier un petit personnage dans un labyrinthe tout en évitant les ennemis, les pièges et en collectant un maximum de pièces d'or.

Voici les étapes que nous allons suivre :

1. **Séance 1 :** Affichage de la bordure du labyrinthe sous forme de caractères ;
2. **Séance 2 :** Saisie de caractères et déplacement du personnage (un 'X') ;
3. **Séance 3 :** Ajout de l'"intelligence" du jeu avec la détection des collisions avec les murs ;
4. **Séance 4 :** Chargement de différents labyrinthes ou niveaux depuis des fichiers ;
5. **Séance 5 :** Ajout des ennemis, des pièges et des trésors. Gestion des combats ;
6. **Séance 6 :** Amélioration de l'interface en mode texte ;
7. **Séance 7 :** Interface graphique.

Pour vous aider, à tout moment vous pourrez vous reportez aux index pour trouver des rappels sur des instructions ou des notions déjà vues.

La première des choses à faire maintenant est de vérifier l'installation sur votre ordinateur, de l'éditeur spécialisé Wing IDE 101 (version multi-plateforme) puis de configurer l'éditeur pour la version 3 de Python.#

1 Séance : Afficher des caractères

Pour notre jeu, nous allons commencer par afficher la bordure du labyrinthe sous la forme de caractères. Cette tâche est simple, mais elle va faire surgir de nombreux problèmes que nous résoudrons en faisant intervenir des notions de programmation particulières.

- Les chaînes de caractères
- Des données dans un tableau
- `print()` pour afficher des données
- Boucler pour ne pas répéter la même chose

Si nous choisissons le caractère # pour indiquer un mur, pour afficher un mur horizontal nous pouvons taper :

```
In [1]: print("####################")

        ####################
```

Attention aux erreurs de syntaxe :

```
In [2]: print("####################


          File "<ipython-input-2-55a9e4695eb8>", line 1
        print("####################
                                    ^
    SyntaxError: EOL while scanning string literal
```

Le message nous indique que la fin de ligne (EOL pour *End Of Line*) a été atteinte alors que la chaîne de caractères (`string`) n'était pas terminée.

Pour ne pas se retrouver constamment confrontés à ce type d'erreur sur les chaînes de caractères, nous devons comprendre comment elles doivent être écrites.

1.1 Les chaînes de caractères

In [4]:
```
print("Il dit : "Coucou, c'est moi!"")
```

```
  File "<ipython-input-4-333423f81340>", line 1
    print("Il dit : "Coucou, c'est moi!"")
                          ^
SyntaxError: invalid syntax
```

Cette erreur est tout à fait prévisible, car elle nous a été signalée avant que nous validions notre code ! Comment ? Grâce aux couleurs qui sont employées pour afficher les caractères que nous tapons. Nous voyons ainsi que `"Il dit :"` est une chaîne, que `Coucou, c` est analysé comme une instruction (qui ne sera bien sûr pas comprise) et que `'est moi""`) est une chaîne qui n'est visiblement pas fermée ... Etrange, car elle ne commence pas par des guillemets. Faisons donc un petit test :

In [5]:
```
print('Ceci est une chaîne de caractères')
```

```
Ceci est une chaîne de caractères
```

Pas d'erreur ! Une chaîne de caractères peut donc être encadrée soit par des guillemts, soit par des apostrophes. Donc si nous voulons afficher le caractère guillemet, le plus simple est d'encadrer notre chaîne de caractères par des apostophes :

In [6]:
```
print('Il dit : "Coucou!"')
```

```
Il dit : "Coucou!"
```

Mais que se passe-t-il si nous souhaitons intégrer à une même chaîne des guillemets et des apostrophes ?

In [7]:
```
print('Il dit : "Coucou, c'est moi!"')
```

```
  File "<ipython-input-7-75e22fbb7db3>", line 1
    print('Il dit : "Coucou, c'est moi!"')
                               ^
SyntaxError: invalid syntax
```

Pour empêcher qu'un caractère guillemet ou apostrophe soit interprété en tant qu'ouverture ou fermeture d'une chaîne de caractères, il faut le faire précéder par un caractère d'échappement \. On dit que l'on *protège* le caractère. Les deux écritures suivantes sont alors équivalentes :

In [9]:
```
print('Il dit : "Coucou, c\'est moi!"')
print("Il dit : \"Coucou, c'est moi!\"")
```

```
Il dit : "Coucou, c'est moi!"
Il dit : "Coucou, c'est moi!"
```

En utilisant ce type d'écriture, il existe des caractères spéciaux qui désignent par exemple la tabulation (\t) ou le retour à la ligne (\n). De ce fait, si vous voulez insérer un anti-slash dans une chaîne de caractères, il faudra lui-même le protéger : \\.

Chaînes de caractères sur plusieurs lignes

En utilisant le retour à la ligne \n, nous pourrions définir le labyrinthe :

In [10]:
```
print("####################\n#    #       #     #\n###      #######   #")
```

```
####################
#    #       #     #
###      #######   #
```

Ca fonctionne ... Mais ce n'est pas très lisible. Pour vraiment pouvoir écrire une chaîne de caractères sur plusieurs lignes, il faut tripler les caractères d'ouverture et de fermeture que sont les guillemets ou les apostrophes. Par exemple avec les guillemets comme caractère d'encadrement :

In [11]:
```
print("""####################
#    #       #     #
###      #######   #""")
```

```
####################
#    #       #     #
###      #######   #
```

C'est déjà mieux, même si les caractères de la première ligne ne sont pas alignés avec les autres.

Stockage dans une variable

Le décalage entre la première ligne et les autres est induit par l'appel à l'instruction `print` et les trois guillemets ouvrant la chaîne de caractères. Mais une chaîne de caractère peut avoir une existence propre complètement indépendante d'un appel à une quelconque instruction :

In [13]:
```
"Une chaine de caracteres"
```

Out [13]:
```
'Une chaine de caracteres'
```

Attention aux caractères accentués, il faudra éventuellement indiquer l'encodage des caractères, en début de script

In [14]:
```
"Une chaîne de caractères"
```

```
Out [14]:
         'Une cha\xc3\xaene de caract\xc3\xa8res'
```

L'interpréteur Python affiche la chaîne de caractères, cette expression est donc valide. Par contre, elle ne fait absolument rien ! Si nous lançons le même code depuis l'éditeur en le sauvegardant (avec l'extension `.py`) puis en l'exécutant(triangle vert ou `Ctrl-Alt-V`), il ne se passera rien du tout.

Pour pouvoir réutiliser cette chaîne, il faut l'enregistrer dans la mémoire de l'ordinateur et pour pouvoir la retrouver il faut lui associer une étiquette, un nom compréhensible par un être humain qui soit différent des noms utilisés par la machine et que l'on appelle *des adresses mémoires*. Lorsque l'on utilise le nom de notre étiquette, nous retrouvons la valeur qui lui est associée. C'est ce qu'on appelle **une variable**.

Imaginez une énorme bibliothèque contenant des milliers et des milliers d'ouvrages. À chacun de ces livres nous associons une étiquette contenant un nom unique (il est impossible de trouver deux fois le même nom sur l'ensemble des étiquettes utilisées dans cette bibliothèque). Pour retrouver un livre, il nous suffit de connaître le texte contenu par l'étiquette qui, en général, nous indiquera en plus la positon du livre. Nous aurons ainsi accès très simplement à l'information stockée à cet emplacement, à savoir le contenu du livre.

Pour aller plus loin, nous pouvons même considérer qu'une nouvelle édition du livre paraît : nous remplaçons l'ancien livre par le nouveau sans changement d'étiquette. Nous avons ainsi modifié le contenu, mais pas la méthode d'accès.

Pour les variables, tout fonctionne exactement comme avec la bibliothèque : on associe une étiquette à une information.

```
In [20]: ma_variable = "Une chaîne de caractères"
```

Attention : lorsque vous choisissez un nom de variable, Python distingue les lettres minuscules des lettres majuscules. On dit qu'il est sensible à la casse. Les variables ma_variable, Ma_variable et MA_VARIABLE sont ainsi trois variables distinctes. On récupère l'information à partir de l'étiquette :

```
In [21]: print(ma_variable)

         Une chaîne de caractères
```

On peut mettre à jour les informations :

```
In [22]: ma_variable = "Des caractères"
         print(ma_variable)

         Des caractères
```

Pour notre labyrinthe, nous pouvons ainsi définir une variable qui contiendra le premier niveau :

```
In [24]: level_1="""\
         ####################
         #    #       #     #
         ###      #######   #"""

         print(level_1)

         ####################
         #    #       #     #
         ###      #######   #
```

En Python, pour indiquer que nous souhaitons aller à la ligne et qu'une instruction n'est pas terminée, il faut utiliser le caractère `anti-slash` (cf. celui ajouter avant notre premier passage à la ligne.

Multiplication de chaînes de caractères

Pour de petits labyrinthes de 20×20 cases, la méthode précédent est déjà fastidieuse ... Imaginez s'il faut créer des labyrinthes 100×100 ! Une des solutions que l'on peut appliquer en Python est de multiplier une chaîne de caractères, celle-ci sera répétée n fois :

```
In [25]: print("coucou "*3)

         coucou coucou coucou
```

Ce mécanisme peut bien sûr être employé avec les variables :

```
In [26]: mur_complet = "#"*100
         print(mur_complet)

         ############################################################################
         ##############################
```

Et si nous souhaitons insérer le contenu d'une variable à l'intérieur d'une chaîne ? Par exemple, nous devrons déplacer un personnage que nous symboliserons par le caractère X :

```
In [27]: perso = "X"
```

Comment faire pour qu'il apparaisse à l'intérieur d'une autre chaîne représentant une partie du labyrinthe ?

Affichage formaté

La première solution est de coller plusieurs chaînes de caractères entre elles pour n'en former plus qu'une. Ce mécanisme est appelé **concaténation** et s'effectue à l'aide du signe $+$:

```
In [29]: "Bonjour, " + "c\'est " + "moi"

Out [29]:
         "Bonjour, c'est moi"
```

L'opérateur $+$ agit donc comme une colle entre les chaînes de caractères. Python sait additionner deux entiers et "coller" des chaînes de caractères ... Par contre, il ne sait pas additionner un entier et une chaîne, ni même les coller :

```
In [30]: "Votre score : " + 1230

         ---------------------------------------------------------------
-------------
    TypeError                                 Traceback (most recent
call last)



        <ipython-input-30-1f6322d827d4> in <module>()
    ----> 1 "Votre score : " + 1230
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

Nous obtenons une erreur TypeError nous indiquant qu'il faudrait convertir l'entier en chaîne de caractères pour que cela fonctionne. En Python, l'instruction `str` permet ce transtypage :

In [31]:
```
str(1230)
```

Out [31]:
```
'1230'
```

Donc pour concaténer un entier et une chaîne, il faut faire appel à `str` :

In [32]:
```
"Votre score : " + str(1230)
```

Out [32]:
```
'Votre score : 1230'
```

Pour insérer le personnage dans le labyrinthe, comme il ne s'agit que de chaînes de caractères et que la concaténation fonctionne de la même manière avec des variables :

In [33]:
```
print("#      " + perso + "             #")
```

```
#      X             #
```

Il existe une autre méthode plus élégante : **le formatage**. Cette technique va consister à créer une chaîne de caractères dans laquelle nous indiquerons à l'aide de symboles spéciaux qu'il y a des "trous", des zones d'insertion qui seront comblées par des valeurs :Je _(1)_en _(2)_ un jeu de labyrinthes.Si nous disons que 1 vaut "programme" et 2 vaut "Python", nous obtenons :

Je programme en Python un jeu de labyrinthes.Python permet de mettre en place exactement cette méthode. Les "trous" du texte seront représentés par {} et pour indiquer les valeurs de remplacement nous utiliserons l'instruction `format` :

In [34]:
```
print("#      {}             #".format(perso))
```

```
#      X             #
```

Le formatage a deux intérêts : il peut servir de modèle et la conversion sous forme de chaîne de caractères des valeurs à insérer se fait de manière complètement automatique.

In [37]:
```
print("Votre {} : {}".format("score", 1230))

modele = "Votre {} : {}"
print(modele.format("score", 1230))
print(modele.format("nombre de vie(s)", "deux"))
```

```
Votre score : 1230
Votre score : 1230
Votre nombre de vie(s) : deux
```

Nous sommes maintenant capables d'afficher le labyrinthe de différentes manières. Pour pouvoir suivre la position du personnage, il serait plus simple de parler de coordonnées (x, y) où x désigne le numéro de ligne et y le numéro de colonne sur la grille du labyrinthe. Ca tombe bien, les chaînes de caractères sont déjà adaptées à ce type de traitement !

Accès par index

Une chaîne de caractères est une suite de caractères : il y a le premier caractère, puis le deuxième, le troisième, etc., jusqu'au dernier caractère. Nous pouvons alors représenter notre chaîne comme un tableau où la première case contient la première lettre et ainsi de suite. Pour repérer plus facilement chaque case, nous allons les numéroter. Un être humain commencerait par 1 ... Un ordinateur commence par 0 (pour différentes raisons très valables). On dit d'un tableau où les cases sont numérotées qu'il est **indexé**.

À partir de ces index, on peut indiquer quel caractère on souhaite afficher :

In [38]:
```
variable = "ma chaîne"
print(variable[0])
print(variable[3])
```

```
m
c
```

Dans le cas du labyrinthe, nous avons ainsi accès aux colonnes... Mais comment faire pour les lignes ?

1.2 Des données dans un tableau

Python nous permet de créer nos propres tableaux, appelés **listes**. Pour définir un tableau, il suffit d'indiquer ses différents éléments entre crochets :

In [39]:
```
tab=["########", "#      #"]
print(tab[0])
```

```
########
```

Le premier élément de notre tableau, `tab[0]`, que nous avons affiché étant une chaîne de caractères, on peut le considérer lui aussi comme un tableau pour n'afficher que le deuxième caractère :

In [40]:
```
print(tab[0][1])
```

```
#
```

Nous pouvons ainsi créer des listes (ou tableaux) à 2, 3 ou n dimensions. Pour l'utilisation que nous allons en faire pour stocker notre labyrinthe de 20×20, nous pouvons définir la liste suivante en raffinant un peu l'affichage : un $-$ représente un mur horizontal, un $|$ représente un mur vertical et un $+$ représente un angle :

In [5]:
```
level_1 = [
"+------------------+",
"|                  |",
```

```
"|                    |",
"|                    |",
"+--------------------+"]
```

La variable `level_1` contient le labyrinthe de premier niveau, il reste maintenant à l'afficher.

1.3 `print` pour afficher des données

Lorsqu'on utilise l'instruction `print`, elle affiche les données qui lui sont passées en paramètre (ce que l'on indique entre parenthèses). Jusqu'à présent, nous avons utilisé cette instruction de façon très basique, mais il va maintenant falloir comprendre comment elle fonctionne car, visiblement, l'affichage d'une liste ne fonctionne pas comme nous l'aurions souhaité :

In [42]:
```
print(level_1)
```

```
['+--------------------+', '|                    |', '|
|', '|                    |', '|                    |', '|
|', '|                    |', '|                    |', '|
|', '|                    |', '|                    |', '|
|', '|                    |', '|                    |', '|
|', '|                    |', '|                    |', '|
|', '|                    |', '+--------------------+']
```

Nous obtenons l'affichage de la liste sous la forme ... d'une liste. Cela semble logique. Mais comment fonctionne donc cette instruction ?

`print` concatène et affiche les données qui lui sont transmises en paramètres entre parenthèses et séparées par des virgules :

In [ ]:
```
>>> print("Ma","chaîne")
Ma chaîne
```

Vous remarquerez que le caractère espace a été ajouté automatiquement entre les deux chaînes de caractères : il s'agit du caractère de séparation par défaut. Pour modifier ce caractère, il faut ajouter un paramètre sous la forme `sep="..."` :

In [ ]:
```
>>>print("Ma","chaîne", sep="___")
Ma___chaîne
```

De la même manière, il est possible de modifier le caractère de fin de ligne qui est par défaut un retour à la ligne \n. Le paramètre à ajouter est alors `end="..."` :

In [ ]:
```
>>>print("Ma","chaîne", sep="___", end="!")
Ma___chaîne!
```

Enregistrez et exécutez le code suivant pour vous en convaincre, vous obtiendrez un affichage sur une seule ligne :

In [ ]:
```
score = 1230
vies = 3
print("Votre", "score :", score, end="\t")
print("Vie(s) :", vies)
```

Tout cela ne résout pas notre problème de liste ... En fait, ce qu'il faudrait faire c'est parcourir toutes les lignes du tableau pour les afficher :

```
In []: print(level_1[0])
       print(level_1[1])
       print(level_1[2])
       ...
       print(level_1[19])
```

Cette fois-ci, nous obtenons un affichage correct. Mais là encore, si le labyrinthe tient sur une grille de 100×100, il va falloir écrire cent lignes de code où une seule valeur changera : le numéro de ligne.

1.4 Boucler pour ne pas répéter la même chose

L'instruction `for` permet de parcourir tous les éléments d'une liste : elle lit le premier élément et le place dans une variable que l'on peut ensuite utiliser, puis elle lit le deuxième élément qu'elle place dans la variable et ainsi de suite tant qu'il y a des éléments dans la liste. Il s'agit d'**une boucle**.

```
In [47]: for tour in [1,2,3,4] :
             print(tour)
```

```
1
2
3
4
```

tour est la variable qui prendra ses valeurs dans la liste suivant le mot-clé `in`. La présence du caractère **:** à la fin de la ligne est très importante. C'est ce caractère qui indique le début d'un bloc : une suite d'instructions qui sont exécutées dans un certain contexte (ici, tant qu'il y a des éléments dans la liste).

Pour bien voir ces instructions, il faut ajouter une indentation - 4 espaces ou une tabulation - en début de ligne. Si vous oubliez cette iondentation, votre code ne fonctionnera pas ou aura une toute autre signification. Le shell Python ajoute automatiquement une indentation après une ligne se terminant par **:**. Pour indiquer que vous avez terminé votre bloc, il faut simplement appuyer sur [Entrée] sur une ligne vierge pour sortir du bloc indenté.

Voici deux exemples de boucles où seule l'indentation diffère :

```
In [50]: for tour in [1,2,3,4] :
             print(tour)

         print("c'est fini !")
         print("------------")
         for tour in [1,2,3,4] :
             print(tour)
             print("c'est fini !")
```

```
1
2
3
4
c'est fini !
------------
1
c'est fini !
2
c'est fini !
3
c'est fini !
4
```

```
c'est fini !
```

Pour afficher notre labyrinthe, il faudra donc parcourir la variable `level_1`. Pour conserver le travail effectué, enregistrez sous forme de programme la définition de la variable `level_1` puis ajoutez les lignes suivantes :

In [51]:
```
for ligne in level_1 :
    print(ligne)
```

```
+--------------------+
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

Pour récapituler :

- Une chaîne de caractères se comporte, en première approximation, comme une liste de caractères ;
- Une liste permet de stocker des éléments qui seront accessibles à partir d'un index ;
- Une variable enregistre en mémoire une information en vue de la ré-utiliser plus tard ;
- L'instruction `print` concatène et traduit automatiquement en chaînes de caractères les éléments qui lui sont transmis sous forme de paramètres ;
- Une boucle répète automatiquement des lignes d'instructions structurées en bloc (décalées par rapport à la ligne débutant le bloc). Cette répétition s'achève par la modification d'une condition telle que "y a-t-il encore deséléments dans la liste ?".

#

2 Séance : Saisir des données

Le personnage doit pouvoir être déplacé dans le labyrinthe. Pour cela, le joueur appuiera sur des touches qui permettront de commander des déplacements ... Mais comment récupérer l'information transmise par l'utilisateur ?L'instruction `print` permet d'afficher des données. La commande que l'on pourrait qualifier de réciproque, permettant d'obtenir des données de l'utilisateur, est l'instruction `input`. Cette commande va afficher un message (que

nous définirons), puis attendra que l'utilisateur saisisse des caractères. La fin de la saisie se fera par un appui sur la touche [Entrée].

Nous demanderons à l'utilisateur de choisir son déplacement à l'aide d'une instruction semblable à la suivante :

In []:
```
>>> input("Quelle direction ? ")
Quelle direction ? Haut
'Haut'
```

Vous pouvez voir qu'après avoir appelé l'instruction, la chaîne de caractères est affichée et le curseur vient se placer après elle, en attenet d'une saisie de l'utilisateur. Lors de l'appui sur la touche [Entrée], la saisie est transmise à la machine (qui nous répond ici qu'il s'agit de la chaîne de caractères 'Haut'). En l'état, nous ne pouvons rien faire de cette information : il faudrait l'enregistrer pour pouvoir y accéder par la suite. Comment ? Grâce aux variables que nous avons vues à la séance 1 : comme l'instruction `input` nous répond qu'elle a bien lu une chaîne de caractères, nous pouvons stocker cette dernière dans une variable :

In []:
```
>>> deplacement = input("Quelle direction ? ")
Quelle direction ? Haut
>>> print(deplacement)
Haut
```

Mais que se passe-t-il si nous saisissons un entier ?

In []:
```
>>> input("Entrez un nombre : ")
Entrez un nombre : 12
'12'
```

L'instruction nous répond qu'elle a lu une chaîne de caractères ... Pour pouvoir faire des calculs avec cette valeur, il va falloir la convertir avec une instruction (qui traduit un élément en entier) : l'instruction `int`. Cette conversion peut être effectuée de deux manières différentes :

• Soit en stockant la saisie de l'utilisateur dans une variable, puis en la convertissant :

In []:
```
>>> val = input("Entrez un nombre : ")
Entrez un nombre : 12
>>> val_entier = int(val)
>>> val_entier
12
```

Ici, **val** est une chaîne de caractères alors que **val_entier** est un entier. Nous aurions également pu remplacer le contenu de **val** (chaîne de caractères) par sa conversion en entier en écrivant :

In []:
```
>>> val = int(val)
```

• Soit en convertissant la saisie de l'utilisateur avant de l'enregistrer dans une variable :

In []:
```
>>> val = int(input("Entrez un nombre : "))
Entrez un nombre : 12
>>> val
12
```

Ici, lorsque l'utilisateur saisit sa réponse, l'instruction `input("Entrez un nombre :")` est remplacée par la saisie, c'est-à-dire la chaîne de caractères `'12'`. On applique alors sur cette chaîne la conversion à l'aide de l'instruction `int` et la variable *val* contient donc un entier.Nous sommes maintenant capables d'interroger, l'utilisateur sur le déplacement qu'il souhaite effectuer. À chaque déplacement, il faudra afficher à nouveau le labyrinthe dans lequel la position du personnage aura été modifiée. Pour pouvoir travailler proprement, il serait intéressant de séparer les différentes parties de notre code : affichage du labyrinthe et du personnage, saisie du déplacement, etc. Tout ceci se fait en créant des fonctions.

2.1 Les fonctions

Une fonction permet de mettre en place une technique de contrôle ancestrale : “Diviser pour régner”. Le code est écrit par petits blocs, ce qui permet d’isoler correctement les différentes parties et de les modifier simplement.Une fonction en informatique reprend le principe des fonctions mathématiques : on donne (éventuellement) une ou des valeur(s) en entrée et on obtient une valeur en sortie. Une fonction qui calcule le double d’un nombre donné s’écrira : $f(x) = 2x$. Si on veut savoir quel est le double de 3, nous appellerons la fonction f en lui indiquant que x vaut 3 : $f(3)$. Le résultat, la valeur de retour de la fonction sera 2×3 soit 6. Commençons donc par écrire cette fonction en Python :

In [1]:
```
def f(x):
    return 2*x
```

La partie de définition du nom de la fonction et des paramètres (valeurs) qui lui seront transmis se fait sur la première ligne par `def f(x):` qui correspond à `f(x)=`. Notez que, comme pour les boucles, nous débutons un nouveau bloc : toutes les instructions qui seront indentées (décalées par rapport à la marge de gauche), feront partie de la fonction.

La valeur de retour de la fonction est définie par l’instruction `return` (ici, ce sera 2*x où * est l’opérateur correspondant à la multiplication ...). Quand nous appellerons cette fonction, la valeur de retour sera calculée, renvoyée à la ligne de code ayant effectué l’appel, puis utilisée. Prenons l’exemple d’un simple affichage :

In []:
```
>>> print("Résultat : ", f(3))
Résultat : 6
```

Avant que l’instruction `print` puisse être exécutée, elle a besoin de savoir à quoi correspond $f(3)$. La fonction f est donc appelée et la valeur de retour est utilisée dans le `print`. C’est comme si nous avions directement écrit :

In []:
```
>>> print("Résultat : ", 6)
Résultat : 6
```

Utilisons maintenant ce mécanisme pour afficher un labyrinthe. À la séance 1, nous avons défini une variable `level_1` contenant uniquement les bordures du labyrinthe (reprenez le code). Une fonction permettant d’afficher n’importe quel labyrinthe acceptera en paramètre une variable contenant le labyrinthe et effectuera la boucle permettant de parcourir tous les éléments :

In [3]:
```
def affiche_labyrinthe(lab):
    for ligne in lab:
        print(ligne)
```

lab représente une liste. Nous appellerons la fonction en utilisant `level_1` en paramètre et ce sera donc la valeur de *lab* :

In [6]:
```
affiche_labyrinthe(level_1)
```

```
+--------------------+
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
```

```
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

Nous obtiendrons bien l'affichage du labyrinthe. Vous aurez sans doute remarqué que la fonction *affiche_labyrinthe* ne contenait pas l'instruction `return`... Donc, elle ne renvoie pas de valeur ? En fait si ! En Python, toute fonction doit renvoyer une valeur. Si, arrivée à la fin d'une fonction, Python ne rencontre pas d'instruction `return`, il l'ajoutera automatiquement et renverra une valeur spéciale indiquant qu'il n'y a rien à renvoyer. Cette valeur, c'est `None` et c'est un peu comme si Python ajoutait en dernière ligne de la fonction `return None`.

Si nous utilisons une variable pour stocker la valeur de retour de *affiche_labyrinthe*, nous obtiendrons l'affichage du labyrinthe, puis notre variable contiendra la valeur `None` :

In [7]:
```
val_retour = affiche_labyrinthe(level_1)
```

```
+--------------------+
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

S'il ne s'agit que de dessiner les contours d'un labyrinthe, nous pouvons exploiter pleinement la puissance des fonctions en paramétrant la taille du labyrinthe :

In [8]:
```
def affiche_bordures_labyrinthe(taille) :
    print("+{}+".format("-"*(taille-2)))   #Utilisation de l'affichage
     #formaté et de la multiplication de chaînes de caractères
    for i in range(taille-2) :
        print("|{}|".format(""*(taille-2))) # Répétition de l'affichage
        # de la ligne (taille -2 fois)
    print("+{}+".format("-"*(taille-2)))
```

Ici, j'ai créé une fonction à laquelle on peut transmettre une valeur, le paramètre *taille*. Cette valeur est utilisée ensuite pour déterminer le nombre de caractères à afficher sur chaque ligne.Dans la boucle, une nouvelle instruction est apparue : la fonction : `range`. Lorsque nous avons abordé les boucles, nous avions vu que l'instruction `for` permet de parcourir l'ensemble des éléments d'une liste. Mais ici, nous n'avons pas une liste, nous avons l'instruction

`range(taille-2)` ... Que fait donc cette instruction ? Dans le shell Python, si nous tapons la ligne suivante nous obtiendrons un début de réponse :

```
In []: >>> range(10)
       range(0,10)
```

La réponse de Python à notre instruction est simplement une autre instruction ... Il ne s'agit donc pas d'une liste. Essayons cette instruction au sein d'une boucle `for` pour si cela fonctionne et ce que l'on peut afficher :

```
In [9]: for i in range(10) :
            print(i)

        0
        1
        2
        3
        4
        5
        6
        7
        8
        9
```

La variable *i* prend des valeurs allant de 0 jusqu'à 9. C'est donc un peu comme si `range(10)` nous avait renvoyé la liste $[0, 1, 2, 3, \ldots, 9]$. D'ailleurs, on peut le vérifier en convertissant `range(10)` en liste :

```
In [10]: list(range(10))

Out [10]:
         [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`range(10)` ne renvoie pas directement une liste pour des raisons d'**optimisaton** de la place occupée en mémoire (si vous créez une liste d'un million d'éléments seulement pour répéter un million de fois un bloc vous utilisez beaucoup de mémoire pour rien !).

La fonction `range` n'admet pas forcément qu'un seul paramètre, on peut définir une plage de nombres depuis un nombre de départ jusqu'à un nombre d'arrivée, en sautant éventuellement des nombres :• `range(fin)` : compte de 0 à `fin-1` ;

• `range(debut,fin)` : compte de `debut` à `fin-1` ;

• `range(debut,fin,saut)` : compte de `debut` à `fin-1` en sautant entre chaque nombre `saut` nombres. Voici un exemple permettant d'obtenir la liste des entiers entre 0 et 10, en comptant de 2 en 2 :

```
In [11]: list(range(0,10,2))

Out [11]:
         [0, 2, 4, 6, 8]
```

Si nous reprenons notre programmedepuis le début, en y ajoutant une fonction permettant de récupérer le choix de déplacement d'un joueur, voici ce que nous obtenons :

```
In []: level_1 = [
       "+------------------+",
       "|                  |",
```

```
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"|                   |",
"+-------------------+"]

def affiche_labyrinthe(lab):
    for ligne in lab:
        print(ligne)

def affiche_bordures_labyrinthe(taille) :
    print("+{}+".format("-"*(taille-2)))
    for i in range(taille-2) :
        print("|{}|".format(" "*(taille-2)))
    print("+{}+".format("-"*(taille-2)))

def choix_joueur():
    return input("Votre déplacement (Haut/Bas/Droite/Gauche) ?")

affiche_bordures_labyrinthe(20)
dep = choix_joueur()
print(dep)
```

Notre programme commence à être conséquent avec plus de quarante lignes et il est bien sûr hors de question d'utiliser le shell Python qui ne nous permettrait pas de modifier proprement notre code.

Mais nous avons déjà un peu de mal à nous relire. Il faudrait que nous puissions ajouter des notes qui nous permettraient de comprendre rapidement à quoi correspond chaque fonction ou chaque partie du programme.

2.2 Des notes en guise d'aide

Il est possible d'ajouter des lignes dans le programme qui ne seront pas interprétées, des sortes de *post-it* invisibles. Un peu comme si l'ordinateur ne pouvait pas les voir mais le développeur si. Pour cela, il existe plusieurs techniques.

Tout d'abord le commentaire dit "en ligne" : tout ce qui suit le caractère # sera considéré comme un commentaire et ne sera pas exécuté :

In [13]:
```
print("Coucou") # Ceci ne sera pas lu
```

```
Coucou
```

In [14]:
```
# print("Coucou")
```

Ce type de commentaires permet d'indiquer ce que fait une ligne. Il peut également être utilisé pour segmenter le code en grande parties. Reprenons le début de notre programme de jeu :

In []:
```
##########
# Définition des différents niveaux
## Niveau 1

level_1 = [
"+--------------------+",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"+--------------------+"]
```

La coloration syntaxique va mettre en exergue les lignes 1 à 3 et nous permettra de naviguer plus simplement dans le code.

Une autre méthode permettant d'ajouter des commentaires est l'utilisation des chaînes de caractères. Nous avions vu qu'une valeur non affectée à une variable ne produisait aucune action particulière en dehors du shell Python. Nous allons utiliser ce mécanisme pour commencer le code :

In []:
```
"**********"
"Définition des différents niveaux"
"Niveau 1"
```

Ceci est valable d'un point de vue syntaxique, vous n'obtiendrez pas de messages d'erreur ... Par contre ce n'est pas la techniuqe préconisée en Python, on préfèrera utilser les commentaires en ligne. Les commentaires écrits à l'aide de chaînes de caractères sont appelés des **docstrings** et, positionnés après la première ligne de définition d'une fonction, ils peuvent être lus par une instruction d'aide qui les affichera avec un formatage spécial. On utilisera les chaînes de caractères multilignes pour obtenir un texte plus lisible :def affiche_labyrinthe(lab): “”" Affichage d'un labyrinthe

```
    lab : Variable contenant le labyrinthe
"""
for ligne in lab:
    print(ligne)
```

Lorsque des fonctions sont commentées de cette sorte, l'instruction `help` permet d'afficher le commentaire lorsque l'on se trouve dans le shell Python. Voici un petit exemple effectué justement dans le shell Python :

In []:
```
>>>def f(x):
    """Calcule le double de la valeur passée en paramètre

    x : Valeur dont on souhaite calculer le double
    """
    return 2*x

>>>help(f)
Help on function f in module __main__:

f(x)
    Calcule le double de la valeur passée en paramètre
```

```
    x : Valeur dont on souhaite calculer le double
```

Pour être utile, un commentaire doit-être pertinent et concis : commentez vos lignes de code ... mais pas trop ! Le mieux est l'ennemi du bien. À vouloir donner trop d'informations vous perdrez le lecteur qui n'y trouvera pas les explications qu'il recherche. De plus, certaines lignes de code parlent d'elles-mêmes.Nous n'avons pas énormément avancé sur le jeu, nous avons surtout appris à structurer le code de manière à ce que nous puissions développer par la suite notre programme le plus rapidement et surtout, avec le moins d'erreurs possible.

Pour récapituler :

- Une **fonction** est un bloc de code qui s'exécute lorsque l'on fait appel à lui par son nom en lui fournissant éventuellement des paramètres qui modifieront son comportement ;
- Un **commentaire** est une chaîne de caractères qui ne représente aucune instruction et qui ne sera pas interprétée lors de l'exécution du programme. Les commentaires ne sont utiles que pour les développeurs, les utilisateurs n'y ayant pas accès.

#

3 Séance : Les tests de déplacements

Une fois que l'utilisateur a saisi un déplacement, il faut le répercuter à l'écran. Et que se passe-t-il si le joueur désire déplacer son personnage dans un mur ? Il va falloir que nous mettions en place des tests pour nous assurer que le déplacement est autorisé.

- Un peu de logique booléenne pour pouvoir tester des valeurs
- Tester des valeurs et effectuer des actions en fonction du résultat obtenu
- Découpage en tranches
- Une première version de notre jeu

Pour positionner notre personnage dans le labyrinthe, nous allons numéroter chaque case en fonction de sa colonne et de sa ligne. La première case qui se trouve en haut et à gauche (celle qui correspond à un caractère + dans `level_1`) se verra donc affecter la valeur $(0, 0)$. La case suivante, à droite de $(0, 0)$, se trouve sur la colonne suivante de la même ligne, donc sur $(1, 0)$. Pour savoir où se trouve le personnage du joueur, nous aurons besoin d'un couple de données (colonne, ligne). Pour stocker ces valeurs qui évoluent au cours de la partie, nous allons utiliser une liste.

Au départ, le personnage se trouvera sur la première case libre du labyrinthe, c'est-à-dire la case $(1, 1)$:

In [1]:
```
perso = "X"
pos_perso = [1,1]
```

Pour savoir sur quelle colonne se trouve le personnage, il faudra consulter `pos_perso[0]` et pour connaître la ligne se sera `pos_perso[1]`.

Comme le labyrinthe est affiché ligne par ligne, il va nous falloir commencer par détecter si nous nous trouvons sur la ligne sur laquelle le personnage est positionné. POur rappel, la fonction permettant d'afficher le labyrinthe est la suivante :

In [ ]:
```
def affiche_labyrinthe(lab):
    """
        Affichage d'un labyrinthe
```

```
        lab : Variable contenant le labyrinthe
    """
    for ligne in lab:
        print(ligne)
```

Si nous voulons savoir sur quelle ligne nous nous trouvons, il faut ajouter **un compteur**. Il s'agira d'une simple variable qui aura pour valeur de départ 0 et à laquelle nous ajouterons une unité à chaque passage dans la boucle :

In []:
```
def affiche_labyrinthe(lab):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
    """
    n_ligne = 0                # on initialise le compteur
    for ligne in lab:
        print(ligne)
        n_ligne = n_ligne +1   # on incrémente le compteur
```

Il faut maintenant être capable de comparer le numéro de ligne courant contenu dans la variable *n_ligne* avec le numéro de ligne du personnage `pos-perso[1]`. Pour cela, nous allons avoir besoin des tests qui font intervenir la **logique booléenne**.

3.1 Un peu de logique booléenne pour pouvoir tester des valeurs

En informatique, tous les mécanismes de tests sont basés sur la logique booléenne :

une expression est soit vraie (**True**) soit fausse (**False**). Les opérateurs de comparaison permettent de comparer deux valeurs et d'indiquer si la condition énoncée est vérifiée (`a` est plus grand que `b`, `a` et `b` ont la même valeur, etc.). Faisons un essai de shell Python :

In []:
```
>>> a = 1
>>> b = 2
>>> a < b
True
>>> a > b
False
```

Il existe sept opérateurs de comparaison :

- $>$: supérieur à ;
- $<$: inférieur à ;
- $>=$: supérieur ou égal à ;
- $<=$: inférieur ou égal à ;
- $!=$: différent de ;
- $==$: égal à ;
- is : identique à.

Comme vous pouvez le constater, l'égalité se teste à l'aide d'un opérateur contenant deux fois le caractère $=$. Pourquoi cette redondance ? Tout simplement pour ne pas confondre un test avec une affectation à une variable. Imaginons que nous reprenions notre variable `a` et que nous voulions tester si la valeur qu'elle contient est bien 5 :

In []:
```
>>> a == 5
```

```
False
>>> print(a)
1
```

La ligne `a == 5` peut se lire “Est-ce que `a` a pour valeur 5 ?”. La réponse est bien sûr non. Par contre, si nous nous trompons et que nous oublions un signe égal :

In []:
```
>>> a = 5
>>> print(a)
5
```

Nous avons modifié la valeur de `a` ! Il faut donc toujours être très prudent et penser à bien mettre l’opérateur de comparaison en doublant le signe = lors d’une comparaison.

Vous vous demandez sans doute pourquoi s’il existe un opérateur d’égalité, un opérateur d’identité a été ajouté ...L’identité est plus forte que l’égalité. Certaines valeurs peuvent-être considérées comme équivalentes à d’autres. Par exemple, 1 est équivalent à `True` et 0 est équivalent à `False` dans un contexte de test. Prenons le cas de la valeur 0 :

In []:
```
>>> 0 == False
True
```

Par contre, l’identité va chercher si, en plus, les deux valeurs comparées sont de même type. Or, ici, nous avons un entier et un booléen :

In []:
```
>>> 0 is False
False
```

Si vous souhaitez effectuer des tests un peu plus complexes, vous aurez besoin de “combiner” des tests. Cette succession de tests se fait à l’aide de trois opérateurs booléens :

• *and* : deux conditions doivent être vérifiées. Par exemple, `a == 1 and b < 3` signifie que pour que l’expression soit vraie, `a` doit valoir 1 et `b` doit contenir un nombre inférieur à 3 ;

• *or* : au moins une des deux conditions doit être vérifiée. Par exemple, `a == 1 or b < 3` signifie que pour que l’expression soit vraie, soit `a` doit valoir 1, soit `b` doit contenir un nombre inférieur à 3, soit `a` vaut 1 et `b` est inférieur à 3 ;

• *not* : la condition ne doit pas être vérifiée ; si elle est fausse, on la considère comme vraie et réciproquement.

De ces opérateurs on tire les tables de vérités, tableaux indiquant la valeur de retour des tests auxquels on applique des opérateurs booléens. Ici, 1 représente `True` et 0 représente `False`.

A	B	A `and` B	A `or` B	`not` A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Nous savons tester des conditions, mais il faut également être capable d’exécuter certaines instructions si ces conditions sont vraies et d’autres si elles sont fausses.

3.2 Tester des valeurs et effectuer des actions en fonction du résultat obtenu

Pour créer un “aiguillage” permettant d’exécuter un bloc d’instructions *A* si une condition est respectée et un bloc d’instructions *B* sinon, nous allons utiliser la commande `if`. Si la condition qui suit cette instruction est vraie, alors le bloc suivant est exécuté :

In []:
```
>>> a = 1
>>>if a == 1 :
       print("La valeur est bien 1")

La valeur est bien 1
```

La ligne de test peut se lire : “Si `a` est égal à 1 alors exécute le bloc”. Si la variable `a` est différente de 1, il ne se passera rien. On peut ajouter un traitement pour ce cas grâce à l’instruction `else` :

In []:
```
>>>if a == 2 :
       print("La valeur est bien 2")
   else :
       print("Valeur différente de 2 :", a)

Valeur différente de 2 : 1
```

Ici, `a` est différent de 2 donc c’est le deuxième bloc qui est affiché. On peut lire ce code de la manière suivante : “Si `a` est égal à 2 alors exécute le premier bloc, sinon exécute le second bloc”.

Notez l’indentation des lignes (décalage par rapport à la marge), dans un programme nous aurions un alignement parfait :

In []:
```
if a == 2 :
    print("La valeur est bien 2")
else :
    print("Valeur différente de 2 :", a)
```

Nous sommes maintenant capables de détecter la ligne sur laquelle afficher le personnage dans notre labyrinthe. Pour rappel, la position du personnage est stockée dans la liste `pos_perso`.

`pos_perso[0]` donne le numéro de la colonne et `pos_perso[1]` donne le numéro de ligne :

In []:
```
#paramètre pos_perso pour la position du personnage sous forme de liste
def affiche_labyrinthe(lab, pos_perso):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
    """
    n_ligne = 0
    for ligne in lab:
        if pos_perso[1] == n_ligne :
            # ne s'exécute que si le test est vrai
            print(ligne, "<- ligne du personnage")
        else :
            # ne s'exécute que si le test est faux
            print(ligne)
        # l'incrémentation s'exécute toujours (voir indentation)
        n_ligne = n_ligne +1
```

À l’exécution, nous obtenons quelque chose comme :

In []:
```
+--------------------+
|                    | <- ligne du personnage
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
```

```
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

Pour pouvoir vraiment afficher le personnage, il faut lire la ligne caractère à caractère de manière à remplacer le caractère en position `pos_perso[0]` (la colonne ou se trouve le personnage) par le caractère représentant le personnage. Il ne s'agit en fait que d'une boucle sur l'ensemble des caractères de la chaîne. Pour pouvoir la réaliser, il faut seulement savoir que l'instruction `len` nous donne le nombre de caractères que contient une chaîne :

In []:
```
>>> chaine = "|                   |"
>>> perso = "X"
>>> pos_perso = [1,1]
>>> for i in range(len(chaine)) :
        if i == pos_perso[0]:
            print(perso, end = "")
        else :
            print(chaine[i], end = "")

|X                  |
```

L'instruction la plus difficile à lire ici est `range(len(chaine))`. Pour mieux la comprendre il faut la décortiquer étape par étape. Tout d'abord, `len(chaine)` nous donne la taille de la chaîne soit 20. L'instruction globale est donc équivalente à `range(20)` et nous savons que cette commande produit une sorte de liste allant de 0 à 19. Donc, nous passerons vingt fois dans la boucle où la variable *i* variera de 0 à 19.

Si *i* correspond à la position du personnage nous l'affichons et sinon, nous affichons le ième caractère de la chaîne de caractères, c'est-à-dire `chaine[i]`. Notez l'utilisation du paramètre `end` dans l'instruction `print` pour ne pas avoir de retour à la ligne intempestif (sinon nous afficherions notre ligne sous la forme d'une colonne . . .).

Nous pourrions incorporer ce morceau de code dans notre fonction *affiche_labyrinthe*, mais il existe une manière plus simple d'afficher le personnage.

3.3 Découpage en tranches

Si nous pouvions découper une chaîne de caractères en "tranches", comme du saucisson, comment ferions-nous pour y placer notre personnage ? Il faudrait conserver tous les caractères avant la position `pos_perso[0]`, insérer le caractère du personnage et afficher tous les caractères de la ligne du labyrinthe se trouvant après la position `pos_perso[0]`. En Python, un mécanisme appelé "découpage en tranches" - ou **slicing** - permet de sélectionner des portions à l'intérieur d'une chaîne de caractères.

Pour cela, au lieu d'indiquer entre crochets seulement l'indice du caractère que l'on souhaite récupérer, nous indiquerons la position du caractère de départ, celle du caractère de fin et éventuellemnrt un pas (en fait, exactement comme les paramètres de la fonction `range`, mais au lieu de séparer les paramètres par des virgules, nous utiliserons le caractère :). Tous ces paramètres ne sont pas obligatoires, on peut indiquer :

• Seulement la position du caractère de départ : la portion se terminera alors à la fin de la chaîne de caractères initiale ;

• La positon du caractère dedépart et de fin : la portion contiendra donc tous les caractères désignés ;

• La position du caractère de départ, de fin et un pas : la portion contiendra alors les caractères de la chaîne initiale obtenus en la parcourant par sauts d'un nombre de caractères indiqués par le pas.

Prenons quelques exemples :

In []:
```
>>> chaine = "ABCDEFGHIJKLM"
>>> print("Taille de la chaîne :", len(chaine))
Taille de la chaîne : 13

>>> chaine[0:5]
'ABCDE'
```

Ici, nous sélectionnons la portion de chaîne commençant en position 0 jusqu'à la position 5 non comprise.

In []:
```
>>> chaine[0:5:2]
'ACE'
```

En prenant la même chaîne que précédemment mais en modifiant le pas, nous ne récupérons plusqu'une lettre sur deux.

In []:
```
>>> chaine[:5]
'ABCDE'
```

Si nous omettons le premier paramètre, la valeur par défaut 0 est utilisée. L'écriture précédente est donc équivalente à `chaine[0:5]`. Pour être plus précis, on utilise également ici le pas par défaut qui a pour valeur 1. L'écriture équivalente complète de notre instruction est donc `chaine[0:5:1]`.

In []:
```
>>> chaine[6:]
'GHIJKLM'
```

Cette fois, c'est la position finale qui est la valeur par défaut. Notez qu'une autre écriture permet d'accéder au dernier caractère avec `chaine[6:len(chaine)]` (`len(chaine)` renvoie le nombre total de caractères, soit la position du dernier caractère +1). L'instruction `chaine[6:]` est équivalente à `chaine[6::1]` puisque le pas de 1 est la valeur par défaut.

In []:
```
>>> chaine[::2]
'ACEGIKM'
```

Pour finir, nous récupèrons une lettre sur deux sur l'ensemble de la chaîne (les valeurs par défaut de début et de fin de chaîne nous amènent à considérer la chaîne dans son intégralité).Si nous adoptons le slicing sur notre labyrinthe pour placer le personnage, en fonction du numéro de la colonne où se trouve le personnage, la ligne commencera par son contenu normal jusqu'à la colonne du personnage (`ligne[0:pos_perso[0]]`), nous ajouterons ensuite le personnage (*perso*), puis les caractères suivants du labyrinthe (`ligne[pos_perso[0]+1:]`).

3.4 Une première version de notre jeu

Avec toutes les connaissances que nous avons acquises jusqu'à maintenant, nous pouvons produire une première version de notre jeu où il sera possible de voir le personnage se déplacer en fonction des choix du joueur. Pour cela, nous allons un peu modifier la fonction de saisie de l'action du joueur :

In []:
```
##########
# Définition des différents niveaux
## Niveau 1
level_1 = [
"+------------------+",
"|                  |",
"|                  |",
"|                  |",
"|                  |",
```

```
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"+--------------------+"]


def affiche_labyrinthe(lab, perso, pos_perso):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage
                    [ligne, colonne]

        Pas de valeur de retour
    """
    n_ligne = 0
    for ligne in lab:
        if  n_ligne == pos_perso[1]:
            #slicing
            print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0]+1:])
        else :
            print(ligne)
        n_ligne = n_ligne +1
```

Jusqu'ici rien de nouveau, j'ai seulement utilisé le slicing pour placer le personnage dans le labyrinthe en fonction de la position indiquée par la liste `pos_perso` qui est passée en paramètre de la fonction en même temps que le labyrinthe et le caractère représentant le personnage.

Nous allons modifier notre fonction de saisie des déplacements de manière à ce qu'elle interroge une autre fonction pour savoir si le déplacement est valide et éviter que le personnage ne rentre dans un mur par exemple). Nous appellerons cette fonction *verification_deplacement* : (*le code suivant est à mettre à la suite du précédent vers la ligne 47*)

In []:
```
def verification_deplacement(lab, pos_col, pos_ligne):
    """
        Indique si le déplacement du personnage est autorisé ou pas.

        lab : Labyrinthe
        pos_ligne : position du personnage sur les lignes
        pos_col : position du personnage sur les colonnes

        Valeurs de retour :
          None : déplacement interdit
          [col, ligne] : déplacement autorisé sur la case indiquée
                                    par la liste
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire
    # de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
```

```
        pos_col > (n_cols -1) :
            #le symbole \ indique que la ligne n'est pas terminée
        return None
    elif lab[pos_ligne][pos_col] != " " :
        return None
    else :
        return [pos_col, pos_ligne]
```

Si le déplacement n'est pas autorisé (en dehors de l'aire de jeu ou sur un mur), la fonction renvoie la valeur spéciale `None` (rien). Sinon, elle renvoie une liste contenant les nouvelles coordonnées du personnage dans le labyrinthe.

La fonction qui permet au joueur de saisir son choix de déplacement va maintenant faire appel à la fonction *verification_deplacement* pour s'assurer que le déplacement est autorisé. Un choix supplémentaire sera proposé : "Quitter" pour arrêter le programme et sortir du jeu. (le code suivant est à mettre à la suite du précédent vers la ligne 72)

In []:
```
def choix_joueur(lab, pos_perso):
    """
        Demande au joueur de saisir son déplacement
                    et vérifie s'il est possible.
        Si ce n'est pas le cas affiche un message,
                    sinon modifie la position du perso
                    dans la liste pos_perso

        lab : Labyrinthe
        pos_perso : liste contenant la position du personnage
                    [colonne, ligne]

        Pas de valeur de retour
    """
    choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
    if choix == "H" or choix == "Haut" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] -1)
    elif choix == "B" or choix == "Bas" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] +1)
    elif choix == "G" or choix == "Gauche" :
        dep = verification_deplacement(lab, pos_perso[0] -1, pos_perso[1])
    elif choix == "D" or choix == "Droite" :
        dep = verification_deplacement(lab, pos_perso[0] +1, pos_perso[1])
    elif choix == "Q" or choix == "Quitter" :
        os._exit(1)
        # attention il faut importer le module os
        # en début de script il faut écrire : import os

    if dep == None :
        print("Déplacement impossible")
    else :
        pos_perso[0] = dep[0]   # modification du contenu de la liste
        pos_perso[1] = dep[1]   # pos_perso
```

Ici, quelques explications supplémentaires sont nécessaires à propos du paramètre `pos_perso`. En effet, la variable qui est transmise à la fonction *choix_joueur* va voir sa valeur modifiée, or ce n'est pas le comportement normal comme le montre l'exemple suivant :

In []:
```
>>> a =1
>>> def modifier(param) :
        param = param +1
        print("Nouvelle valeur :", param)

>>> print(a)
1
>>> modifier(a)
```

```
Nouvelle valeur : 2
>>> print(a)
1
```

Comme vous le voyez, la valeur de la variable `a` est transmise à la fonction *modifier* et copiée dans la variable `param`. Nous avons donc deux variables distinctes. On parle de **passage de paramètres par valeur**. En Python, ce mécanisme s'applique à tous les types de base : entier, réel, booléen, etc.

En ce qui concerne les listes, le passage de paramètres se fait **par adresse** : il s'agit du même emplacement mémoire qui est utilisé, il n'y a donc pas copie de la valeur. C'est comme si l'on travaillait sur la même variable :

In []:
```
>>> l = [1,2,3]
>>> def modifier(liste) :
        liste[0]=0
        print("Nouvelle liste :", liste)

>>> print(l)
[1,2,3]
>>> modifier(l)
Nouvelle liste : [0,2,3]
>>> print(l)
[0,2,3]
```

Tout ceci nopus amène à aborder la visibilité des variables : une variable **locale** est une variable définie dans un bloc et détruite en sortie de bloc.

In []:
```
>>> def ma_fonction() :
        variable = 2
        print(variable)

>>> ma_fonction()
2
>>> print(variable)
Retraçage (dernier appel le plus récent) :
  Fichier "<string>", ligne 1, dans <fragment>
builtins.NameError: name 'variable' is not defined
```

Ici, `variable` n'existe que dans le bloc de définition de la fonction *ma_fonction*.

Réciproquement, une variable **globale** est une variable définie dans le bloc principal du programme (contre la marge de gauche) et qui est visible dans n'importe quel bloc :

In []:
```
>>> ma_variable = "Coucou"
>>> def ma_fonction() :
        print(ma_variable)

>>> ma_fonction()
Coucou
```

Comme vous le voyez, la variable `ma_variable` n'est pas passée en paramètre de *ma_fonction* ... et pourtant elle est accessible. **Il est préférable d'éviter d'utiliser les variables globales pour ne pas subir des effets de bord** (variables modifiées on ne sait où dans le programme). Ces erreurssont très difficile à retrouver et à corriger et donc particulièrement chronophages.

Pour revenir à notre programme, il nous manque une fonction qui va gérer le jeu : afficher le labyrinthe, demander au joueur où il souhaite déplacer son personnage et recommencer. (le code suivant est à mettre à la suite du précédent vers la ligne 102)

In []:
```
def jeu(level, perso, pos_perso):
    """
        Boucle principale du jeu. Affiche le labyrinthe dans ses différents
```

```
        états après les déplacements du joueur.

        level : Labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage
                                        [colonne, ligne]
    """
    while True :
        affiche_labyrinthe(level, perso, pos_perso)
        choix_joueur(level, pos_perso)
```

J'ai utilisé une nouvelle forme de boucle : la boucle `while` qui signifie "tant que". Tant que la condition est vraie, nous exécutons le code du bloc qui est associé à la boucle.

Ici, la boucle est infinie puisque `True` ne changera jamais de valeur. La seulr façon de quitter le programme est de taper "Quitter" lors du choix de déplacement.

Il ne nous rest plus qu'à utiliser les fonctions que nous avons définies au sein d'un programme. C'est ce que font les dernières lignes suivantes :

In []:
```
##########
# Programme principal
# Initialisation du personnage
perso = "X"
pos_perso = [1,1]
# Lancement de la partie
jeu(level_1, perso, pos_perso)
```

Vous pouvez maintenant "jouer" à déplacer le personnage dans le labyrinthe.

Pour récapituler :

- Une **condition** est une expression dont l'évaluation donne une valeur booléenne qui peut valoir `True` ou `False` ;
- Pour effectuer un branchement logique en fonction d'une condition, on utilise l'instruction `if` suivie éventuellement d'un traitement `else`. Pour enchaîner les tests, on peut utiliser `elif` qui est une forme contractée de `else if` ;
- Le **slicing** permet de découper une portion d'une chaîne de caractères en spécifiant son début, sa fin et le pas de déplacement ;
- Les listes sont passées par adresse dans les fonctions : une modification de leur valeur à l'intérieur de la fonction modifie la valeur de la variable passée en paramètre ;
- Une **variable locale** est une variable qui n'existe qu'à l'intérieur d'un bloc défini ;
- L'instruction **while** permet de créer une boucle ou un bloc sera exécuté tant qu'une condition ne sera pas invalidée.

#

4 Séance : De nouveaux labyrinthes

RetourNous pouvons déplacer un personnage dans un grand cadre vide. Il est temps d'ajouter des niveaux à notre jeu !

- Chargement des labyrinthes depuis des fichiers externes
- Changements de niveaux
- Ajout de fonctionnalités à Python et structuration du code

Nous avons choisi de stocker les labyrinthes sous forme de listes, comme le montre l'exemple du premier niveau :

In []:
```
level_1 = [
"+--------------------+",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"+--------------------+"]
```

Pour commencer, nous allons **optimiser notre code**. Une liste est une structure à laquelle on peut accéder en lecture et en écriture (on peut modifier un élément). Dans le cas du labyrinthe, les murs ne vont certainement pas bouger ! Nous pouvons alors utiliser une autre structure, s'utilisant exactement comme une liste : **le tuple**.

Un tuple est une liste optimisée pour l'accès en lecture et qui n'est pas modifiable … En fait, c'est le principe de fonctionnement des chaînes de caractères. Pour définir un tuple, on n'utilise plus les crochets mais les parenthèses :

In [1]:
```
t=(1, 2, 3)
print(t[0])
```

```
1
```

Bien sûr, à partir du moment où vous tenterez de modifier le contenu d'un tuple vous obtiendrez un message d'erreur :

In [2]:
```
t[0]=5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-2-e36dcdb8325f> in <module>()
```

```
----> 1 t[0]=5


TypeError: 'tuple' object does not support item assignment
```

La modification de notre premier niveau sera donc minime : remplacer les crochets par des parenthèses. Tout le rest du code continuera de fonctionner comme avant puisque nous ne modifions pas cette liste.

In []:
```
level_1 = (
"+--------------------+",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"+--------------------+")
```

Dans notre première verson du jeu, il manque beaucoup de choses pour pouvoir réellement jouer, mais un élément essentiel est absent : la sortie du labyrinthe ! Nous allons la symboliser par le caractère **O** :

In []:
```
level_1 = (
"+--------------------+",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                    |",
"|                   O|",
"+--------------------+")
```

Pour pouvoir détecter la présence du personnage sur la sortie, nous devons modifier la fonction de vérification des déplacements :

In []:
```
def verification_deplacement(lab, pos_col, pos_ligne):
    """
        Indique si le déplacement du personnage est autorisé ou pas.

        lab : Labyrinthe
        pos_ligne : position du personnage sur les lignes
        pos_col : position du personnage sur les colonnes

        Valeurs de retour :
          None : déplacement interdit
          [col, ligne] : déplacement autorisé
                                    sur la case indiquée par la liste
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire
    # de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
       pos_col > (n_cols -1) :
        return None
    elif lab[pos_ligne][pos_col] == "O" :                        #######
        # Une position hors labyrinthe indique la victoire       ##ici##
        return[-1, -1]                                           #######
    elif lab[pos_ligne][pos_col] != " " :
        return None
    else :
        return [pos_col, pos_ligne]
```

Il faut également modifier la boucle du jeu de manière à détecter la victoire dans un niveau :

In []:
```
def jeu(level, perso, pos_perso):
    """
        Boucle principale du jeu. Affiche le labyrinthe dans ses différents
        états après les déplacements du joueur.

        level : Labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage
                                      [colonne, ligne]
    """
    while True :
        affiche_labyrinthe(level, perso, pos_perso)
        choix_joueur(level, pos_perso)
        if pos_perso == [-1, -1] :                               #######
            print("Vous avez passé le niveau !")                 ##ici##
            break                                                #######
```

Nous introduisons ici deux nouveaautés :

• Un test conditionnel entre une variable et une liste. Plutôt que de tester chaque élément de la liste par `pos_perso[0] == -1 and pos_perso[1] == -1`, il est possible de tester les éléments par identification (est-ce que le premier élément de la liste de gauche est égal au premier élément de la liste de droite, etc.) ;

• L'instruction `break`, qui permet de sortir de la boucle. Quelle que soit l'instruction de boucle (`for` ou `while`), l'instruction `break` indiquera au programme qu'il faut interrompre le traitement du bloc courant (bloc relatif à la boucle) et exécuter les commandes suivantes.

Nous pouvons maintenant sortir d'un labyrinthe … Mais si nous passions à un mécanisme un peu plus sophistiqué de gestion des niveaux ? Un mécanisme qui permettrait de créer simplement un nouveau labyrinthe dans un fichier qui serait ensuite chargé par le jeu. Pour cela, il va falloir être capable de lire un fichier texte.

4.1 Chargement des labyrinthes depuis des fichiers externes

Commençons par le plus simple : le format des labyrinthes contenus dans les fichiers. Nous utilisions jusqu'alors des listes de chaînes de caractères, donc nous allons rester dans le même format c'est-à-dire des chaînes de caractères dans un fichier texte. Nous nommerons nos fichiers `level_numero_du_niveau.txt` où `numero_du_niveau` sera un entier identifiant le niveau de manière unique. Le fichier `level_1.txt` pourra par exemple contenir :

In []:
```
+------------------+
|       |          |
|--- + |   -----+ |
|     |         | |
| +  +------+   | |
| |         |   | |
| +---+     +---+ |
|      |           |
|--+  +-----------|
|      |           |
|--+   |   +       |
|   |  |   |   +---|
|   |      |       |
|   +------+---+   |
|              |   |
|    +----+   +   |
|    |             |
| +--|   +---------|
|    |           O|
+------------------+
```

Pour afficher ce niveau, il faut lire le fichier, placer les données dans un tuple et exécuter le code que nous avons déjà écrit. La manipulation d'un fichier passe par trois étapes :

1. Ouverture du fichier : création d'une variable qui nous permettra d'accéder au fichier. Un fichier peut-être ouvert dans différents modes en fonction de ce que l'on souhaite réaliser :
   - en lecture (`read` en anglais) : pour lire des données. Si le fichier que l'on désire ouvrir n'existe pas, un message d'erreur apparaîtra ;
   - en écriture (`write` en anglais) : pour écrire des données. Si le fichier que l'on veut créer existe déjà, il est détruit et remplacé par le nouveau (on dit qu'il est écrasé);
   - en ajout (`append` en anglais) : pour ajouter des données à un fichier existant. Si le fichier à ouvrir existe, l'écriture des données commence à la fin du fichier, il n'y a donc pas de perte d'informations qui étaient précédemment stockées à l'intérieur. Si le fichier n'existe pas, il est créé.
2. Action sur le fichier : en fonction des cas, il s'agira de la lecture ou de l'écriture.
3. Fermeture du fichier : après avoir travaillé sur le fichier, nous indiquons au système que nous n'avons plus besoin d'y acccéder.

D'un point de vue syntaxique, l'ouverture d'un fichier se fait par la fonction `open` :

In []:
```
>>> fic = open("level_1.txt", "r")
```

La variable que nous avons appelée `fic` est un descripteur de fichier. Dorénavant, c'est grâce à elle que nous pourrons accéder au fichier dont nous avons passé le nom en paramètre, c'est-à-dire `level_1.txt`.

Le second paramètre, la chaîne de caractères ne contenant qu'une seule lettre, est le mode d'ouverture du fichier. Nous retrouvons les trois modes décrits précédemment : **"r"** pour `read` - la lecture, **"w"** pour `write` - l'écriture, et **"a"** pour `append` - l'ajout. Il faut savoir que, par défaut, le fichier sera recherché dans le répertoire courant du programme (vous pouvez bien sûr également indiqué un chemin absolu, c'est-à-dire un chemin complet partant de la racine de l'arborescence des fichiers représentée par **/**).

Il existe ensuite plusieurs fonctions permettant de lire des données dans un fichier. Nous considérerons qu'une ligne est une suite de caractères se terminant par un saut à la ligne :

- `readlines()` : lit toutes les lignes du fichier et les place dans une liste ;
- `readline()` : lit une ligne et la place dans une chaîne de caractères ;
- `read()` : lit toutes les données du fichier et les place dans une seule chaîne de caractères ;
- `read(n)` : lit `n` caractères et les place dans une chaîne decaractères.

Dans tous les cas, le saut de ligne qui aura été lu dans le fichier sera conservé dans la chaîne de caractères stockée en mémoire. Donc, si vous l'affichez à l'aide d'un `print` sans désactiver le retour à la ligne automatique du paramètre `end`, vous obtiendrez une ligne vide superflue.

Pour l'écriture il n'y a qu'une seule fonction, `write`, qui prend en paramètre la chaîne de caractères à écrire dans le fichier.

Ces fonctions s'exécutent d'une manière un peu spéciale : on ne leur transmet pas le descripteur de fichier en paramètre, mais on dit qu'elles s'appliquent au descripteur de fichier. Cette nuance fait que l'on ne doit pas écrire : `level = readlines(fic)` mais plutôt :

In []:
```
level = fic.readlines()
```

Vous l'aurez compris, dans le cas de notre jeu c'est la fonction `readlines` qui est la plus adaptée. Iol faudra seulement convertir la liste de résultat en tuple grâce à la fonction de conversion du même nom :

In []:
```
level =tuple(fic.readlines())
```

Une fois que nous avons fini de travailler sur le fichier, nous l'indiquons en fermant le descripteur de fichier :

In []:
```
fic.close()
```

Avant de mettre en place cette technique dans notre jeu, effectuons quelques tests dans le shell Python. Nous allons créer un fichier, y écrire quelques lignes, le refermer puis l'ouvrir pour lire son contenu et s'assurer qu'il correspond bien à ce que nous y avons enregistré. Première étape, la création de fichier :

In []:
```
>>> fic.close()
>>> fic = open("fichier_test.txt", "w")
>>> fic.write("Une ligne...\n")
13
>>> fic.write("Encore une ligne !\n")
19
>>> fic.close()
```

À chaque fois que nous écrivons dans un fichier, la fonction `write` nous indique le nombre de caractères qui ont été écrits. Notez la présence de \n à la fin des chaînes de caracères pour passer à la ligne (sinon nos deux chaînes se trouveront l'une à la suite de l'autre dans le fichier).

La deuxième étape est la lecture. Nous allons réaliser le même type de lecture que ce dont nous allons avoir besoin dans notre jeu :

In []:
```
>>> fic = open("fichier_test.txt", "r")
>>> data = tuple(fic.readlines())
>>> fic.close()
>>> data
('Une ligne...\n', 'Encore une ligne !\n')
```

La variable `data` est un tuple de chaînes de caractères... Mais nous allons être ennuyés par la présence du caractère de retour à la ligne si nous ne voulons pas trop modifier le code que nous avons déjà écrit pour afficher les labyrinthes :

In []:
```
>>> print(data[0])
Une ligne...

>>>
```

Nous affichons bien deux sauts à la ligne ! Une solution pourraît être d'employer le slicing pour supprimer le dernier caractère :

In []:
```
>>> print(data[0][:len(data[0])-1])
Une ligne...
>>>
```

Mais il faudrait quand même modifier le code d'affichage ... En fait, il nous faudrait une instruction capable d'enlever les caractères superflus en fin de chaîne (espace, tabulation, saut à la ligne). Cette instruction existe, c'est `strip`, qui permettra de retirer tous les caractères invisibles du début et de la fin d'une chaîne (`lstrip` pour seulement le début et `rstrip` pour seulement la fin). Là encore, cette instruction s'utilise d'une manière un peu spéciale en l'appliquant directement à une chaîne de caractères :

In []:
```
>>> print(data[0].strip())
Une ligne...
```

Nous pouvons maintenant écrire la fonction de lecture d'un labyrinthe dans un fichier extérieur et supprimer la définition de la variable `level_1` qui nous est désormais inutile :

In []:
```
def charge_labyrinthe(nom) :
    """
        Charge le labyrinthe depuis le fichier nom.txt

        nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)

        Valeur de retour :
            Tuple contenant les données du labyrinthe
    """
    fic = open(nom + ".txt", "r")
    # Lecture des données dans le fichier
    data = fic.readlines()
    fic.close()
    # Parcours de la liste pour supprimer les caractères invisibles
    for i in range(len(data)) :
        data[i] = data[i].strip()

    return tuple(data)     # Conversion de la liste data en tuple
```

La seule autre modification se situe alors dans le programme principal, où il faut lire le niveau avant de lancer le jeu :

In []:
```
##########
# Programme principal
# Initialisation du personnage
perso = "X"
pos_perso = [1,1]
# Lancement de la partie
# Lecture du fichier level_1.txt et stockage sous forme de tuple
level_1 = charge_labyrinthe("level_1")
jeu(level_1, perso, pos_perso)
```

Nous pouvons maintenant jouer... à un niveau ! Il est temps d'ajouter des labyrinthes.

4.2 Changements de niveaux

Nous avons mis en place un mécanisme permettant de définir les labyrinthes dans des fichiers distincts. Il est donc simple de changer de niveau en fonction du nombre de niveaux disponibles. Nous avons décidé de nommer nos fichiers `level_num.txt` et donc, s'il y a vingt niveau de disponibles, il faudra charger les fichiers `level_1.txt` à `level_20.txt` ...

Une simple boucle dans le programme principal peut faire cela :

In []:
```
##########
# Programme principal
# Initialisation du personnage
perso = "X"
pos_perso = [1,1]
n_levels = 20  # Variable contenant le nombre total de niveaux
# Lancement de la partie
 # Boucle où i prendra ses valeurs entre 1 et n_levels
for i in range(1, n_levels +1) :
    # On concatène la chaîne "level_" et le contenu de i transtypé par str()
    level = charge_labyrinthe("level_" + str(i))
    jeu(level, perso, pos_perso)
```

Il est ainsi possible de changer de niveau... Mais le joueur n'a aucune information sur le niveau où il se trouve. Nous allons ajouter une barre de score qui indiquera le niveau. Cette barre sera gérée par une fonction indépendante :

In [2]:
```
def barre_score(n_level) :
    """
        Barre de score affichant les données du jeu

        n_level : niveau courant

        Pas de valeur de retour
    """
    print("Level : {:3d}".format(n_level))
```

Cette fonction est très simple puisqu'elle ne contient en fait qu'une ligne d'affichage. Il faut toutefois noter que nous employons un affichage formaté un peu particulier, puisqu'au lieu de `{}` nous avons employé `{:3d}`. Le code `:3d` donné entre les accolades permet d'indiquer que nous souhaitons afficher un entier (la lettre `d`) sur trois chiffres (le chiffre `3`) alignés à droite. Cette astuce nous assure qu'il n'y aura pas de décalage d'affichage au passage des dizaines ou des centaines.

Comme c'est la fonction *jeu* qui est chargée de l'affichage du labyrinthe, c'est elle aussi qui va afficher la barre de score et il va falloir lui transmettre un paramètre supplémentaire : le numéro du niveau courant. L'appel de la fonction va donc être modifié :

In []:
```
# Lancement de la partie
for n_level in range(1, n_levels_total +1) :
    level = charge_labyrinthe("level_" + str(n_level))
    jeu(level, n_level, perso, pos_perso)
```

Et bien sûr la fonction *jeu* elle-même devra être revue :

In []:
```
def jeu(level, n_level, perso, pos_perso):
    """
        Boucle principale du jeu. Affiche le labyrinthe dans ses différents
        états après les déplacements du joueur.

        level : Labyrinthe
        n_level : numéro du niveau courant
        perso : caractère représentant le personnage
```

```
        pos_perso : liste contenant la position du personnage
                                    [colonne, ligne]
    """
    while True :
        affiche_labyrinthe(level, perso, pos_perso)
        barre_score(n_level)
        choix_joueur(level, pos_perso)
        if pos_perso == [-1, -1] :
            print("Vous avez passé le niveau !")
            break
```

Nous n'avons pas vu ici de nouvelle instruction ou de nouvelle technique de programmation, nous avons simplement amélioré le jeu. Pour continuer les améliorations, nous allons maintenant avoir besoin d'une nouvelle notion permettant d'ajouter des fonctionnalités au langage.

4.3 Ajout de fonctionnalités à Pyhton et structuration du code

Il est possible d'enrichir le langage Python avec de nouvelles fonctionnalités. Cela passe par des sortes de bibliothèques de fonctions, que l'on peut charger et utiliser à loisir : **les modules**. Et Python en compte des milliers, dont certains sont installés par défaut avec toute distribution !

Les instructions fournies par un module sont inconnues tant que celui-ci n'a pas été chargé (c'est lui qui contient le code qui définit ces instructions). Pour charger un module, on utilise l'instruction `import` suivie du nom du module. Toutes les fonctions fournies par le module sont alors accessibles en préfixant leur nom par le nom du module. Prenons l'exemple du module `sys`, voici ce qui se passe :

In [6]:
```
>>> print(platform)
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)


        <ipython-input-6-6c876b381dda> in <module>()
    ----> 1 print(platform)



        NameError: name 'platform' is not defined
```

In [7]:
```
>>> import sys
>>> print(platform)
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)
```

```
        <ipython-input-7-12fd1d8fedf4> in <module>()
          1 import sys
    ----> 2 print(platform)



        NameError: name 'platform' is not defined
```

Après chargement du module, il ne faut pas oublier de préfixer la variable par le nom du module :

In [5]:
```
>>> import sys
>>> print(sys.platform)
```

```
linux2
```

Nous sommes donc en mesure de déterminer sous quel système d'exploitation notre jeu est exécuté. Cela va nous permettre de réaliser un appel système grâce à la fonction `system` du module `os` et d'effacer l'écran avant chaque affichage pour avoir l'impression que le personnage se déplace dans le labyrinthe. Ce n'est pas une obligation, mais pour plus de lisibilité, les modules sont généralement chargés en début de programme :

In [7]:
```
import sys
import os
```

La fonction d'effacement de l'écran ne contiendra qu'un test pour savoir quel appel système exécuter :

In [8]:
```
def efface_ecran() :
    """
        Efface l'écran de la console
    """
    if sys.platform.startswith("win") :
        # Si système Windows
        os.system("cls")
    else :
        # Si système Linux ou OS X
        os.system("clear")
```

Cette fonction est appelée dans la fonction *jeu* juste avant d'afficher le labyrinthe :

In []:
```
def jeu(level, n_level, perso, pos_perso):
    """
...
    """
    while True :
        efface_ecran()
        affiche_labyrinthe(level, perso, pos_perso)
...
```

Notre jeu commence à ressembler à un vrai jeu (en mode console, certes).

Arrivés à ce point, nous allons une nouvelle fois restructurer notre code à l'aide des connaissances acquises aujourd'hui. Comme il va y avoir de nouveaux fichiers et que cela risque d'être compliqué à suivre, je vais détailler les mécanismes de la restructuration sur des exemples simples et vous pourrez retrouver l'intégralité du code du jeu en fin de séance.

La restructuration va consister à créer un module contenant les fonctions et ce module sera chargé par le programme principal. Un module, nous l'avons vu est un simple fichier Python. Ainsi, nous pouvons créer le fichier `mon_module.py` :

In []:
```
def ma_fonction() :
    print("OK")

print("Je suis dans mon_module")
```

Si vous vous placez dans le shell Python pour charger ce module, vous aurez :

In []:
```
>>> import mon_module
Je suis dans mon_module
```

Nous ne voulions pas du `print`, nous voulions simplement avoir accès à la fonction que nous proposait le module ! Un test permet alors de savoir si le fichier Python a été appelé de manière directe (comme exécutable par exemple) ou chargé en tant que module :

In []:
```
def ma_fonction() :
    print("OK")

if __name__ == "__main__" :
    print("Je suis dans mon_module")
    print(__name__)
```

La variable `__name__` est une variable spéciale qui prend pour valeur "`__main__`" si le module est exécuté directement, ou le nom du module si celui-ci est chargé.

Exécutons notre code :

In []:
```
Je suis dans mon_module
__main__
```

Et maintenant chargeons-le dans le shell Python :

In []:
```
>>> import mon_module
>>> mon_module.ma_fonction()
OK
```

Le test conditionnel sur la variable `__name__` permet donc de déterminer quelle est la portion de code qui correspond au code principal.

Pour récapituler :

- Un **tuple** est une liste protégée contre l'écriture et qui est plus rapidement accessible ;
- Le traitement d'un fichier passe par trois étapes : ouverture (création du descripteur de fichier), action et fermeture ;
- Il existe trois modes d'ouverture d'un fichier : en écriture **"w"**, en lecture **"r"** et en ajout **"a"** ;
- Un **module** est un fichier qui fournit de nouvelles fonctionnalités ;
- La variable spéciale **__name__** permet de savoir si l'on a chargé un fichier Python en tant que module, ou s'il a été directement exécuté.

Résumé code :

Le fichier `Lab.py` contient les fonctions principales du jeu :

In []:
```
import sys
import os


def charge_labyrinthe(nom) :
    """
        Charge le labyrinthe depuis le fichier nom.txt

        nom : nom du fichier contenant le labyrinthe(sans l'extension .txt)

        Valeur de retour :
            Tuple contenant les données du labyrinthe
    """
    fic = open(nom + ".txt", "r")
    data = fic.readlines()
    fic.close()

    for i in range(len(data)) :
        data[i] = data[i].strip()

    return tuple(data)


def barre_score(n_level) :
    """
        Barre de score affichant les données du jeu

        n_level : niveau courant

        Pas de valeur de retour
    """
    print("Level : {:3d}".format(n_level))


def affiche_labyrinthe(lab, perso, pos_perso):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage
                                        [ligne, colonne]

        Pas de valeur de retour
    """
    n_ligne = 0
    for ligne in lab:
        if  n_ligne == pos_perso[1]:
            print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0]+1:])  #slicing
        else :
            print(ligne)
        n_ligne += 1


def efface_ecran() :
    """
        Efface l'écran de la console
    """
    if sys.platform.startswith("win") :
        # Si système Windows
        os.system("cls")
```

```
    else :
        # Si système Linux ou OS X
        os.system("clear")


def verification_deplacement(lab, pos_col, pos_ligne):
    """
        Indique si le déplacement du personnage est autorisé ou pas.

        lab : Labyrinthe
        pos_ligne : position du personnage sur les lignes
        pos_col : position du personnage sur les colonnes

        Valeurs de retour :
          None : déplacement interdit
          [col, ligne] : déplacement autorisé
                                    sur la case indiquée par la liste
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire
    # de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
       pos_col > (n_cols -1) :
        return None
    elif lab[pos_ligne][pos_col] == "O" :
        # Une position hors labyrinthe indique la victoire
        return[-1, -1]
    elif lab[pos_ligne][pos_col] != " " :
        return None
    else :
        return [pos_col, pos_ligne]



def choix_joueur(lab, pos_perso):
    """
        Demande au joueur de saisir son déplacement
                    et vérifie s'il est possible.
        Si ce n'est pas le cas affiche un message,
        sinon modifie la position du perso dans la liste pos_perso

        lab : Labyrinthe
        pos_perso : liste contenant la position du personnage
                                      [colonne, ligne]

        Pas de valeur de retour
    """
    choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
    if choix == "H" or choix == "Haut" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] -1)
    elif choix == "B" or choix == "Bas" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] +1)
    elif choix == "G" or choix == "Gauche" :
        dep = verification_deplacement(lab, pos_perso[0] -1, pos_perso[1])
    elif choix == "D" or choix == "Droite" :
        dep = verification_deplacement(lab, pos_perso[0] +1, pos_perso[1])
    elif choix == "Q" or choix == "Quitter" :
        os._exit(1)
    if dep == None :
        print("Déplacement impossible")
        input("Appuyez sur <Return> pour continuer")
    else :
        pos_perso[0] = dep[0]
        pos_perso[1] = dep[1]
```

```
def jeu(level, n_level, perso, pos_perso):
    """
        Boucle principale du jeu. Affiche le labyrinthe dans ses différents
        états après les déplacements du joueur.

        level : Labyrinthe
        n_level : numéro du niveau courant
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage
                                        [colonne, ligne]
    """
    while True :
        efface_ecran()
        affiche_labyrinthe(level, perso, pos_perso)
        barre_score(n_level)
        choix_joueur(level, pos_perso)
        if pos_perso == [-1, -1] :
            print("Vous avez passé le niveau !")
            input("Appuyez sur <Return> pour continuer")
            break
```

Le fichier `Jeu_lab.py` est le fichier principal qui va utiliser le module `Lab` pour lancer le jeu. La première ligne permet d'indiquer sous Linux quel programme doit être utilisé pour exécuter le code. Il suffit ainsi de rendre le fichier éxecutable pour lancer le jeu.

In []:
```
#!/usr/bin/python3

import Lab

if __name__ == "__main__" :
    # Initialisation du personnage
    perso = "X"
    pos_perso = [1,1]
    n_levels_total = 20
    # Lancement de la partie
    for n_level in range(1, n_levels_total +1) :
        level = Lab.charge_labyrinthe("level_" + str(n_level))
        Lab.jeu(level, n_level, perso, pos_perso)
    print("Vous avez gagné !")
```

#

5 Séance : Ennemis, combats et trésors

Les bases de notre jeu sont posées et tout semble fonctionner correctement. Mais il reste bien sûr encore des choses à améliorer d'un point de vue technique et du point de vue du jeu, car trouver seulement la sortie d'un labyrinthe manque cruellement de piquant...

- La gestion des erreurs
- Les trésors
- Les ennemis et les combats

Lorsqu'on développe un programme, il y a une partie essentielle à ne pas négliger : la gestion des erreurs. On ne peut pas laisser l'utilisateur complètement perdu face à un écran affichant un message incompréhensible. Imaginez-vous en train de jouer et puis, brusquement, le jeu s'arrête et l'écran prend une couleur uniforme (bleu par exemple, sans aucune arrière-pensée). Vous conviendrez qu'à part le développeur (et encore), personne ne peut comprendre pourquoi le programme s'est arrêté ! L'utilisateur ne sait donc pas si c'est une fausse manipulation de sa part qui à conduit à l'arrêt du jeu, ou un problème plus profond. Quoi qu'il en soit, ce genre de cas ne doit pas apparaître, il faut analyser

les erreurs dans le programme et soit faire en sorte que le programme soit fermé proprement avec un message clair en indiquant la raison, soit permettre à l'utilisateur de corriger l'action ayant conduit à l'erreur.

5.1 La gestion des erreurs

Dans notre jeu, nous avons un exemple flagrant d'erreur non traitée : la saisie du joueur pour déplacer son personnage. En effet, nous ne traitons que les cas où la saisie est correcte ("Haut", "H", "Droite", etc.). Que se passe-t-il si le joueur indique un déplacement qui n'est pas traité ?

In []:
```
| +--|   +---------|
|    |           O|
+-----------------+
Level :   1
Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? a
Retraçage (dernier appel le plus récent) :
  Fichier "./Jeu_lab.py", ligne 13, dans <module>
  Fichier "/ISN/Jeu/Lab.py", ligne 139, dans jeu
    choix_joueur(level, pos_perso)
  Fichier "/ISN/Jeu/Lab.py", ligne 117, dans choix_joueur
    if dep == None :
builtins.UnboundLocalError: local variable 'dep' referenced before assignment
```

Ici, nous avons beaucoup de détails sur la cause de l'erreur, des informations qui peuvent nous servir à corriger le problème… Ce n'est certainement pas à l'utilisateur de le faire ! Notre erreur se situe dans la fonction *choix_joueur* du fichier `Lab.py` en ligne 117 : nous testons le contenu d'une variable qui n'existe pas. Le message d'erreur se lit du bas vers le haut : en bas, la ligne ayant provoqué l'erreur et au-dessus les appels ayant conduit à cette erreur (fonction *jeu* dans `Jeu_lab.py` qui appelle *choix_joueur* dans `Lab.py`). Retrouvons donc le code ayant provoqué l'erreur :

In []:
```
def choix_joueur(lab, pos_perso):
    """
...
    """
    choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
    if choix == "H" or choix == "Haut" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] -1)
    elif choix == "B" or choix == "Bas" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] +1)
    elif choix == "G" or choix == "Gauche" :
        dep = verification_deplacement(lab, pos_perso[0] -1, pos_perso[1])
    elif choix == "D" or choix == "Droite" :
        dep = verification_deplacement(lab, pos_perso[0] +1, pos_perso[1])
    elif choix == "Q" or choix == "Quitter" :
        os._exit(1)
    if dep == None :
        print("Déplacement impossible")
        input("Appuyez sur <Return> pour continuer")
    else :
        pos_perso[0] = dep[0]
        pos_perso[1] = dep[1]
```

Effectivement, si le choix de l'utilisateur est différent de l'un des choix prédéfinis, la variable `dep` n'existe pas puisque nous ne sommes rentrés dans aucun choix conditionnel. Si cette variable existe et contient la valeur `None`, alors nous affichons "Déplacement impossible". Il suffirait donc que la variable `dep` soit égale à `None` pour que, lors d'un choix non traité, nous affichions un message indiquant que l'action ne peut être réalisée et que nous restions dans le programme. Ajoutons donc une initialisation de cette variable :

In []:
```
def choix_joueur(lab, pos_perso):
    """
...
    """
```

```
    dep = None
    choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
```

Cette fois, si nous saisissons une valeur non admise, nous obtenons un message nous indiquant que le déplacement est impossible et nous pouvons poursuivre notre partie :

In []:
```
| +--|   +---------|
|    |           O|
+------------------+
Level :   1
Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? a
Déplacement impossible
Appuyez sur <Return> pour continuer
```

L'analyse que nous venons d'effectuer est une séquence de débogage : nous sommes partis d'un problème précis, nous avons cherché à le comprendre pour le résoudre. Dans tout développement informatique vous rencontrerez de nombreuses erreurs (les fameux bugs) qu'il faudra résoudre. Ici , de plus, l'erreur est provoquée par une saisie particulière de l'utilisateur et sans cette saisie, le bug n'est pas détectable... D'où l'intérêt de gérer tous les cas possibles lors d'une saisie : les choix corrects et ceux qui ne le sont pas.

Il existe un autre type d'erreur, qu'il faut traiter différemment : **les exceptions**. Lorsque vous essayez de convertir sous forme d'entier une chaîne de caractères ne contenant pas un entier, ou lorsque vous cherchez à ouvrir un fichier qui n'existe pas, vous déclenchez un mécanisme d'erreur particulier, une exception :

In [4]:
```
>>> int("Jeu")
```

```
        ---------------------------------------------------------------
-------------
    ValueError                                Traceback (most recent
call last)



        <ipython-input-4-3c450f60fa02> in <module>()
    ----> 1 int("Jeu")



        ValueError: invalid literal for int() with base 10: 'Jeu'
```

Ce que nous appelions jusqu'alors le nom de l'erreur es ten fait le nom de l'exception et c'est grâce à ce nom nous allons pouvoir traiter l'erreur. Dans le cadre de notre jeu, c'est l'ouverture des fichiers de niveaux qui risque de poser problème. Nous allons donc utiliser cet exemple pour mettre en place la gestion des exceptions. La première étape consiste à connaître le nom de l'exception à traiter. Le plus simple est de provoquer une erreur de ce type dans le shell Python :

In [3]:
```
>>> fic = open("mon_fichier", "r")
```

```
        ---------------------------------------------------------------
-------------
    IOError                                   Traceback (most recent
call last)


        <ipython-input-3-cbf2424dffcc> in <module>()
    ----> 1 fic = open("mon_fichier", "r")



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

Il s'agit ici d'une exception **IOError**. Le traitement des exceptions se fait sous la forme d'une structure composée de plusieurs blocs :

• Le premier bloc contient le code à scruter : si une exception est déclenchée dans ce bloc, le programme ne sera pas directement interrompu, mais l'on cherchera dans les blocs suivants s'il y a des instructions permettant de gérer ce cas ;

• Les blocs suivants indiquent comment traiter les différentes exceptions.

D'un point de vue syntaxique, le premier bloc est ouvert par l'instruction `try` ("essaye" en anglais) et les autres blocs par `except` suivi du nom de l'erreur à traiter. Reprenons l'exemple de l'ouverture d'un fichier :

In []:
```
try :
    fic = open("mon_fichier", "r")
except IOError :
    print("Impossible d'ouvrir le fichier !")
    os._exit(1)
```

Si le fichier n'est pas accessible, alors les deux dernières lignes précédentes sont exécutées. Sinon, si tout se déroule correctement, alors le programme continuera son exécution. Si d'autres exceptions peuvent apparaître dans le bloc "scruté", il suffit d'ajouter des blocs de traitement :

In []:
```
erreur = 0

try :
    fic = open("mon_fichier", "r")
except IOError :
    print("Impossible d'ouvrir le fichier !")
    os._exit(1)
except ValueError :
    print("Erreur lors de la conversion d'un entier")
    erreur = erreur +1

print("On continue l'exécution !")
```

L'erreur sur l'ouverture du fichier est bloquante (puisque l'on sort du programme), alors que l'erreur sur la conversion en entier incrémente une variable et poursuit le traitement.

Vous pourriez être tenté d'utiliser ce mécanisme pour empêcher une erreur d'interrompre le programme sans pour autant la traiter. N'en faites surtout rien : vous pourriez engendrer des erreurs plus importantes et votre programme deviendrait très compliqué à déboguer.

5.2 Les trésors

Nous allons ajouter des trésors dans le jeu. Ces trésors seront répartis en trois catégories et seront représentés par un caractère distinct sur la carte chargée en mémoire (l'utilisateur ne verra lui que le même symbole) :

•1 : rapporte entre 1 et 5 pièces d'or ;

•2 : rapporte entre 5 et 10 pièces d'or ;

•3 : rapporte entre 0 et 25 pièces d'or.

Modifions donc notre carte `level_1.txt` :

In []:
```
+------------------+
|       |          |
|--- + |   -----+ |
|    |          | |
| +  +------+   | |
| |         | 1 | |
| +---+     +---+ |
|     |            |
|--+  +------------|
|     |            |
|--+  |   +        |
| 2|  |   |   +----|
|  |  |       |    |
|  +------+---+    |
|             |    |
|    +----+   +    |
| 3  |             |
| +--|   +---------|
|    |           O|
+------------------+
```

Quelles vont être les conséquences de ces modifications ?

• Il va falloir tenir un compte du nombre de pièces d'or du joueur et les afficher dans la barre des scores ;

• Le personnage devra pouvoir se déplacer sur une case trésor et il faudra ensuite supprimer ce trésor ;

• Il faudra être capable de tirer au hasard un entier entre deux entiers donnés pour déterminer la quantité d'or gagnée.

Commençons par le plus simple : le tirage aléatoire. Python dispose d'un module baptisé **random**, qui contient de nombreuses fonctions en relation avec les nombres aléatoires. Pour obtenir une description de ce que font ces fonctions, vous pouvez utiliser le système d'aide pydoc dans le shell Python :

In []:
```
>>> import random
>>> help(random)
Help on module random:

NAME
    random - Random variable generators.
...
```

Bien sûr le descriptif est un peu long et si vous ne savez pas vraiment ce que vous cherchez cela risque de prendre un peu de temps. La fonction `dir` viendra peut-être à votre secours : elle affichera toutes les fonctions liées à un module. En fonction du nom, qui est bien souvent explicite, vous pourrez focaliser ensuite la demande d'aide sur une fonction particulière :

In [9]:
```
>>> import random
>>> dir(random)
```

```
Out [9]:
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType',
'__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
'_acos', '_ceil', '_cos', '_e', '_exp', '_hashlib', '_hexlify', '_inst', '_log',
'_pi', '_random', '_sin', '_sqrt', '_test', '_test_generator', '_urandom',
'_warn', 'betavariate', 'choice', 'division', 'expovariate', 'gammavariate',
'gauss', 'getrandbits', 'getstate', 'jumpahead', 'lognormvariate',
'normalvariate', 'paretovariate', 'randint', 'random', 'randrange', 'sample',
'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
```

Les fonctions dont le nom est précédé de __ sont des fonctions internes au module et qui ne nous concernent pas. En regardant dans la liste suivante, la fonction `randint` devrait faire ce que nous cherchons : `rand` - aléatoire et `int` - entier. En appelant la fonction `help`, nous en obtenons la confirmation.

```
In [11]: >>> help(random.randint)

Help on method randint in module random:

randint(self, a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Donc, pour un trésor de première catégorie délivrant de une à cinq pièces d'or, nous aurons besoin de l'instruction suivante :

```
In [12]: >>> random.randint(1,5)

Out [12]:
2

In [17]: >>> random.randint(1,5)

Out [17]:
3
```

Le principe d'un générateur aléatoire est de délivrer un nombre aléatoire. Donc, si quittez le shell Python et que vous le relancez, `randint` vous fournira d'autres suites d'entiers. Sur un ordinateur, les nombres ne sont pas générés réellement de façon aléatoire, un calcul est effectué pour obtenir ces nombres et il se base sur une valeur de départ. Si l'on fixe cette valeur de départ, les séries de nombres aléatoires seront toujours les mêmes... Ce qui est très utile pendant la phase de mise au point d'un programme. Pour initialiser cette valeur, appelée graine, on utilise la fonction `seed` à laquelle on transmet un paramètre :

```
In [18]: >>> import random
         >>> random.seed(1)
         >>> random.randint(1,5)

Out [18]:
1
```

In [19]:
```
>>> random.randint(1,5)
```

Out [19]:
```
5
```

Si l'on relance le shell Python et que l'on exécute à nouveau ces lignes, nous obtiendrons les mêmes valeurs.

Nous pouvons déterminer le contenu d'un trésor, il faut maintenant que le joueur puisse stocker cet or. Pour cela, nous allons utiliser une variable et nous afficherons la valeur grâce à la fonction *barre_score*. Cette fonction devra donc avoir accès au numéro du niveau courant (`n_level`), à la quantité de pièces d'or du joueur, et par la suite au nombre de points de vie du joueur.

Pour simplifier le traitement des données, nous allons les regrouper au sein d'une structure de liste un peu partculière, dans laquelle on n'accède pas aux éléments à l'aide de leur position, mais à l'aide d'un mot-clé. Cette structure s'appelle **un dictionnaire** et on utilise des accolades pour le créer :

In []:
```
>>> d = { "po" : 0, "pv" : "aucun" }
>>> d
{'pv': 'aucun', 'po': 0}
>>> print(d["pv"])
aucun
>>> d["test"] = True
>>> d
{'test': True, 'pv': 'aucun', 'po': 0}
```

Dans le fichier `Jeu_lab.py`, il faudra donc initialiser un dictionnaire contenant les données du jeu et le transmettre à la fonction *jeu* qui le transmettra elle-même à la fonction *barre_score* (voir le code complet en fin de séance).

In []:
```
import Lab

if __name__ == "__main__" :
    # Initialisation du personnage
    perso = "X"
    pos_perso = [1,1]
    n_levels_total = 20
    data = {
            "or" : 0,
            "pv" : 25,
            "level" : None
            }
    # Lancement de la partie
    for n_level in range(1, n_levels_total +1) :
        level = Lab.charge_labyrinthe("level_" + str(n_level))
        data["level"] = n_level
        Lab.jeu(level, data, perso, pos_perso)
    print("Vous avez gagné !")
```

En ce qui concerne la gestion des trésors, il faut les afficher tous de la même manière. L'affichage du labyrinthe a lieu dans la fonction *affiche_labyrinthe* et c'est donc dans celle-ci que nous allons remplacer toutes les occurences de 1, 2, 3 par # (caractère choisi pour représenter un trésor). Pour cela, nous aloons utiliser la fonction `replace` qui s'applique à une chaîne de caractères et remplace toutes les occurences de son premier paramètre par le second :

In []:
```
def affiche_labyrinthe(lab, perso, pos_perso, tresor):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage [ligne, colonne]
        tresor : caractère représentant le trésor
```

```
        Pas de valeur de retour
    """
    n_ligne = 0
    for ligne in lab:
        for i in range(1, 4) :
            # remplace les chiffres représentant des trésors
            ligne = ligne.replace(str(i), tresor)
        if  n_ligne == pos_perso[1]:
            print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0]+1:])  #slicing
        else :
            print(ligne)
        n_ligne += 1
```

Il faut ensuite que le personnage puisse se retrouver sur une case du trésor et que dans ce cas la quantité d'or soit augmentée du nombre de pièces du trésor. De plus, il faut supprimer l'emplacement du trésor qui a été trouvé... Et là, nous allons avoir un problème : nous avons décidé d'utiliser un tuple pour stocker le labyrinthe et un tuple est une structure non modifiable... Il y a plusieurs solutions pour résoudre ce problème :

1. Nous nous sommes trompés dans la structure de notre programme et il faut repasser le stockage des labyrinthes sous forme de listes ;
2. Il faut modifier le stockage de la position des trésors en les enregistrant, par exemple, sous forme d'une liste de coordonnées. Ce stockage sera alors décorrélé du labyrinthe ;
3. Utiliser la solution 2 tout en conservant les informations de positionnement des trésors dans le labyrinthe. C'est lors de la lecture que nous extrairons les positions des trésors et que nous les stockerons dans une liste ;
4. Utiliser une bidouille en convertissant le tuple en liste, en modifiant la liste, puis en reconvertissant la liste en tuple et en transmettant cet élément en paramètre à toutes les fonctions ;
5. Arrêter tout et pleurer ...

Dans le développement d'un programme, on essaye de penser à tout... Mais on n'y arrive pas toujours. Il faut alors accepter de modifier le code pour que sa structure soit stable et qu'il puisse continuer à évoluer.

Ici, il faudrait appliquer la solution 3, qui est la plus pérenne et évolutive, mais la solution 1 est la plus simple à mettre en oeuvre et c'est celle-ci que nous allons appliquer (la structuration sous forme de tuple du labyrinthe n'était qu'un prétexte pour aborder les tuples...).

Il suffit alors de modifier la fonction *charge_labyrinthe* pour qu'elle renvoie une liste et non plus un tuple et surtout de modifier la fonction *verification_deplacement* :

In []:
```
def verification_deplacement(lab, pos_col, pos_ligne, data):
    """
...
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire
    # de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
       pos_col > (n_cols -1) :
        return None
    elif lab[pos_ligne][pos_col] == "O" :
        # Une position hors labyrinthe indique la victoire
        return[-1, -1]
    elif lab[pos_ligne][pos_col] == "1" or lab[pos_ligne][pos_col] == "2" or \
         lab[pos_ligne][pos_col] == "3" :
        # teste si le personnage se déplace sur un trésor
        # Découverte d'un trésor
        # fonction qui calcule le montant du butin
        decouverte_tresor(lab[pos_ligne][pos_col], data)
        # On supprime le trésor découvert
        lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
```

```
                            lab[pos_ligne][pos_col +1:]
        return [pos_col, pos_ligne]
    elif lab[pos_ligne][pos_col] != " " :
        return None
    else :
        return [pos_col, pos_ligne]
```

Le calcul du montant du butin est délégué à une fonction qui modifiera la variable contenant toutes les informations du jeu :

In []:
```
def decouverte_tresor(categorie, data) :
    """
        Incrémente le nombre de pièces d'or du joueur en fonction du trésor

        categorie : type de trésor
        - 1 : entre 1 et 5 po
        - 2 : entre 5 et 10 po
        - 3 : entre 0 et 25 po
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    if categorie == "1" :
        data["po"] = data["po"] + random.randint(1, 5)
    elif categorie == "2" :
        data["po"] = data["po"] + random.randint(5, 10)
    else :
        data["po"] = data["po"] + random.randint(0, 25)
```

5.3 Les ennemis et les combats

La gestion des ennemis sera très similaire à celle des trésors (pour ne pas alourdir notre code, les ennemis resteront sattiques). La gestion des combats se fera par tirage aléatoire d'un nombre entre 1 et 10 :

•1 : l'ennemi est tué, mais perte de 5 à 10 points de vie ;

•2 à 4 : l'ennemi est tué, mais perte de 1 à 5 points de vie ;

•5 à 10 : l'ennemi est tué.

Les ennemis seront représentés par un caractère $. Nous ne verrons ici que la fonction réglant le combat, le reste des modificatons étant identiques à celles effectuées pour ajouter les trésors (voir code complet en fin de séance).

In [25]:
```
def combat(data) :
    """
        Détermine le nombre de points de vie perdus lors d'un combat

        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    de = random.randint(1, 10)    # Tirage du dé entre 1 et 10
    if de == 1 :
        data["pv"] = data["pv"] - random.randint(5, 10)
    elif de >= 2 and de <= 4 :
        data["pv"] = data["pv"] - random.randint(1, 5)
```

Le jeu dispose de règles très simples, mais il est maintenant parfaitement jouable !

Pour récapituler :

- Il faut toujours gérer toutes les erreurs ;

- Le traitement **try/except** permet d'intercepter et de traiter des exceptions ;
- Pour rechercher de l'aide dans le shell Python on peut utiliser les commandes **help** et **dir** ;
- Un dictionnaire est une liste dans laquelle les éléments ne sont pas indexés par une valeur numérique, mais par une clé qui peut être de n'importe quel type.

Résumé code :

Le fichier `Lab.py` contient les fonctions principlales du jeu. C'est lui qui a été le plus modifié par rapport à la séance précédente :

In []:
```
import sys
import os
import random


def charge_labyrinthe(nom) :
    """
        Charge le labyrinthe depuis le fichier nom.txt

        nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)

        Valeur de retour :
            - une liste avec les données du labyrinthe
    """
    try :
        fic = open(nom + ".txt", "r")
        data = fic.readlines()
        fic.close()
    except IOError :
        print("Impossible de lire le fichier {}.txt".format(nom))
        os._exit(1)

    for i in range(len(data)) :
        data[i] = data[i].strip()

    return data


def barre_score(data) :
    """
        Barre de score affichant les données du jeu

        data : dictionnaire de données de la barre de score

        Pas de valeur de retour
    """
    print("PV : {:2d}      PO : {:4d}      Level : {:3d}".format(data["pv"],
                                                                 data["po"], data["level"]))


def affiche_labyrinthe(lab, perso, pos_perso, tresor):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage [ligne, colonne]
        tresor : caractère représentant le trésor

        Pas de valeur de retour
```

```
    """
    n_ligne = 0
    for ligne in lab:
        for i in range(1, 4) :
            # remplace les chiffres représentant des trésors
            ligne = ligne.replace(str(i), tresor)
        if  n_ligne == pos_perso[1]:
            print(ligne[0:pos_perso[0]] + perso + ligne[pos_perso[0]+1:])  #slicing
        else :
            print(ligne)
        n_ligne += 1


def efface_ecran() :
    """
        Efface l'écran de la console
    """
    if sys.platform.startswith("win") :
        # Si système Windows
        os.system("cls")
    else :
        # Si système Linux ou OS X
        os.system("clear")


def decouverte_tresor(categorie, data) :
    """
        Incrémente le nombre de pièces d'or du joueur en fonction du trésor

        categorie : type de trésor
        - 1 : entre 1 et 5 po
        - 2 : entre 5 et 10 po
        - 3 : entre 0 et 25 po
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    if categorie == "1" :
        data["po"] = data["po"] + random.randint(1, 5)
    elif categorie == "2" :
        data["po"] = data["po"] + random.randint(5, 10)
    else :
        data["po"] = data["po"] + random.randint(0, 25)


def combat(data) :
    """
        Détermine le nombre de points de vie perdus lors d'un combat

        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    de = random.randint(1, 10)    # Tirage du dé entre 1 et 10
    if de == 1 :
        data["pv"] = data["pv"] - random.randint(5, 10)
    elif de >= 2 and de <= 4 :
        data["pv"] = data["pv"] - random.randint(1, 5)


def verification_deplacement(lab, pos_col, pos_ligne, data):
    """
        Indique si le déplacement du personnage est autorisé ou pas.

        lab : Labyrinthe
        pos_ligne : position du personnage sur les lignes
        pos_col : position du personnage sur les colonnes
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)

        Valeurs de retour :
          None : déplacement interdit
```

```
            [col, ligne] : déplacement autorisé sur la case indiquée par la liste
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
       pos_col > (n_cols -1) :
        return None
    elif lab[pos_ligne][pos_col] == "O" :
        # Une position hors labyrinthe indique la victoire
        return [-1, -1]
    elif lab[pos_ligne][pos_col] == "1" or lab[pos_ligne][pos_col] == "2" or \
         lab[pos_ligne][pos_col] == "3" :
        # teste si le personnage se déplace sur un trésor
        # Découverte d'un trésor
        # fonction qui calcule le montant du butin
        decouverte_tresor(lab[pos_ligne][pos_col], data)
        # On supprime le trésor découvert
        lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
                         lab[pos_ligne][pos_col +1:]
        return [pos_col, pos_ligne]
    elif lab[pos_ligne][pos_col] == "$" :
        # Rencontre d'un ennemi
        combat(data)
        lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
                         lab[pos_ligne][pos_col +1:]
        return [pos_col, pos_ligne]
    elif lab[pos_ligne][pos_col] != " " :
        return None
    else :
        return [pos_col, pos_ligne]


def choix_joueur(lab, pos_perso, data):
    """
        Demande au joueur de saisir son déplacement et vérifie s'il est possible.
        Si ce n'est pas le cas affiche un message, sinon modifie la position
        du perso dans la liste pos_perso

        lab : Labyrinthe
        pos_perso : liste contenant la position du personnage [colonne, ligne]
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)

        Pas de valeur de retour
    """
    dep = None
    choix = input("Votre déplacement (Haut/Bas/Droite/Gauche/Quitter) ? ")
    if choix == "H" or choix == "Haut" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] -1, data)
    elif choix == "B" or choix == "Bas" :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] +1, data)
    elif choix == "G" or choix == "Gauche" :
        dep = verification_deplacement(lab, pos_perso[0] -1, pos_perso[1], data)
    elif choix == "D" or choix == "Droite" :
        dep = verification_deplacement(lab, pos_perso[0] +1, pos_perso[1], data)
    elif choix == "Q" or choix == "Quitter" :
        os._exit(1)
    if dep == None :
        print("Déplacement impossible")
        input("Appuyez sur <Return> pour continuer")
    else :
        pos_perso[0] = dep[0]
        pos_perso[1] = dep[1]
```

```
def jeu(level, data, perso, pos_perso, tresor):
    """
        Boucle principale du jeu. Affiche le labyrinthe dans ses différents
        états après les déplacements du joueur.

        level : Labyrinthe
        data : dictionnaire contenant
               - level : le numéro de niveau
               - po    : le nombre de pièces d'or
               - pv    : le nombre de points de vie
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage [colonne, ligne]
        tresor : caractère représentant le trésor
    """
    while True :
        efface_ecran()
        affiche_labyrinthe(level, perso, pos_perso, tresor)
        barre_score(data)
        if data["pv"] <= 0 :
            print("Vous avez PERDU..")
            os._exit(1)
        choix_joueur(level, pos_perso, data)
        if pos_perso == [-1, -1] :
            print("Vous avez passé le niveau !")
            input("Appuyez sur <Return> pour continuer")
            break
```

Le fichier `Jeu_lab.py` n'a été que très peu modifié :

In []:
```
#!/usr/bin/python3

import Lab

if __name__ == "__main__" :
    # Initialisation du personnage
    perso          = "X"
    pos_perso      = [1,1]
    tresor         = "#"
    n_levels_total = 20
    data = {
            "po" : 0,
            "pv" : 25,
            "level" : None
            }
    # Lancement de la partie
    for n_level in range(1, n_levels_total +1) :
        level = Lab.charge_labyrinthe("level_" + str(n_level))
        data["level"] = n_level
        Lab.jeu(level, data, perso, pos_perso, tresor)
    print("Vous avez gagné !")
```

#

6 Séance : Interface console améliorée

Notre jeu est terminé, mais la saisie des déplacements est plutôt laborieuse : chaque saisie doit être validée par un appui sur la touche [Return]. Nous allons donc modifier notre programme pour obtenir une interface plus fluide.

- Utilisation du module curses (sous windows installer cette version du module ou celle qui correspond à votre version de Python ici)
- Le module curses dans notre jeu

Le module **curses** permet de gérer l'affichage dans un terminal et l'interception des événements clavier (dès que vous appuyez sur une touche une information est envoyée au programme). Avant de pouvoir utiliser ce module pour améliorer notre jeu, il faut comprendre comment il fonctionne.

6.1 Utilisation du module curses

L'utilisation du module **curses** impose de passer dans un mode "graphique" particulier. Nous devons donc passer par deux étapes obligatoires : au début du programme il faudra initialiser les paramètres graphiques et à la fin du programme il faudra restaurer les anciennes valeurs. Attention : pour utiliser le module **curses** vous devez forcément exécuter le code depuis un terminal. Sous le shell Python, vous obtiendrez un message d'erreur :

In []:
```
>>> import curses
>>> curses.initscr()
Retraçage (dernier appel le plus récent) :
  Fichier "src/debug/tserver/_sandbox.py", ligne 1, dans <module>
    # Used internally for debug sandbox under external interpreter
  Fichier "/usr/lib/python3.2/curses/__init__.py", ligne 31, dans initscr
    fd=_sys.__stdout__.fileno())
_curses.error: setupterm: could not find terminal
```

Comme nous allons devoir utiliser de nombreuses commandes, il est préférable de débuter directement par un petit programme de test `test_curses.py` :

In []:
```
import curses

if __name__ == "__main__" :
    curses.initscr()       # Initialisation du mode graphique
    curses.noecho()       # Désactivation de l'affichage
                          # des touches tapées au clavier
    curses.cbreak()       # Interception des touches tapées au clavier
                          # (sans appui sur [Return])
    curses.curs_set(0)    # Désactivation de l'affichage du curseur

    window = curses.newwin(40, 79, 0, 0)    # Création d'une fenêtre
    # 40 caractères de haut et 79 de large positionée en (0,0) du terminal
    window.border(0)      # Activation du tracé de la bordure de la fenêtre
    window.keypad(1)      # Activaton du nommage des touches : KEY_UP, etc.
    c = window.getch()    # Attente de l'appui sur une touche du clavier
```

Cet exemple est un peu long pour débuter, mais il faut bien que nous puissions voir quelque chose… Ici, un cadre va apparaître à l'écran et restera affiché jusqu'à ce que l'on appuie sur une touche (n'importe laquelle).

Notez qu'une fois le programme achevé, vous ne voyez plus ce que vous tapez dans le terminal… C'est normal, puisque nous n'avons pas restauré les paramètres d'affichage.

Nous allons donc reprendre notre programme en le structurant à l'aide d'une fonction d'initialisation et une fonction de fermeture. Nous en profiterons également pour ajouter un traitement d'erreur si l'initialisation de la fenêtre graphique est impossible :

In []:
```
import curses

def init_curses(lignes, cols, pos) :
    """
        Initialisation des paramètres graphiques

        lignes : nombre de lignes (en caractères)
        cols   : nombre de colonnes (en caractères)
        pos    : tuple contenant la position du coin supérieur gauche
                 de la fenêtre graphique

        Valeur de retour :
            La fenêtre curses ayant été créée
    """
    curses.initscr()
    curses.noecho()
    curses.cbreak()
    curses.curs_set(0)

    window = curses.newwin(lignes, cols, pos[0], pos[1])
    window.border(0)
    window.keypad(1)
    return window


def close_curses() :
    """
        Restauration des paramètres graphiques
    """
    # Appel des fonctions inverses que celles invoquées dans init_curses
    curses.echo()
    curses.nocbreak()
    curses.curs_set(1)
    curses.endwin()

if __name__ == "__main__" :
    try :
        win = init_curses(40, 79, (0, 0))
        # Attente de l'appui sur une touche
        c = window.getch()
    except curses.error :
        print("Erreur graphique : interruption du programme")
    finally :
        close_curses()
```

Dans cet exemple, nous avons introduit un nouveau bloc lié au traitement de l'exception : le bloc **finally**. Ce bloc est exécuté quoi qu'il se passe, qu'il y ait eu une erreur interceptée ou non. Grâce à ce mécanisme, on s'assure de bien exécuter la fonction de restauration des paramètres initiaux du terminal, même si une erreur est rencontrée.

Pour écrire en couleur, il va falloir activer l'utilisation des couleurs et définir des associations entre un code (que nous utiliserons par la suite) et une paire de couleurs représentant les couleurs de premier plan et d'arrière-plan. Les couleurs que l'on peut utiliser sont stockées sous la forme **curses.COLOR_NAME**, ou **NAME** est le nom d'une couleur en anglais (**RED**, **GREEN**, etc.).

Nous allons définir une fonction d'initialisation des couleurs, ce qui nous permettra ensuite d'afficher des messages colorés. Attention : l'affichage n'est plus exécuté à l'aide de la fonction `print`, mais d'une fonction `addstr` à laquelle on indique en quelle position (x, y) nous souhaitons positionner notre chaîne de caractères.

In []:
```
import curses

def init_curses(lignes, cols, pos) :

...

def close_curses():
```

```
...

def init_colors() :
    """
        Initialisation des couleurs

        Valeur de retour :
            liste contenant le nom des couleurs (index = code couleur)
    """
    curses.start_color()
    curses.init_pair(1, curses.COLOR_RED, curses.COLOR_BLACK)
    curses.init_pair(2, curses.COLOR_GREEN, curses.COLOR_BLACK)
    curses.init_pair(3, curses.COLOR_BLACK, curses.COLOR_BLUE)
    return["RED", "GREEN", "BLUE"]


def color(code, l_color):
    """
        Sélectionne une couleur

        code : nom de la couleur
        l_color : liste des couleurs

        Valeur de retour :
            code de couleur curses
    """
    return curses.color_pair(l_color.index(code) +1)


if __name__ == "__main__" :
    try :
        win = init_curses(40, 79, (0, 0))

        # Initialisation des couleurs
        coul = init_colors()

        # Messages
        win.addstr(1, 1, "Message en rouge", color("RED", coul))
        win.addstr(2, 1, "Message en noir sur fond bleu", color("BLUE", coul))

        # Attente de l'appui sur une touche
        c = window.getch()
    except curses.error :
        print("Erreur graphique : interruption du programme")
    finally :
        close_curses()
```

En valeur de retour de la fonction *color*, nous avons utilisé une nouvelle fonction : **index**. Cette fonction, appliquée à une chaîne de caractères, permet de retrouver l'index correspondant à la valeur qui lui est passée en paramètre (si **RED** se trouve à l'index 0, alors la fonction nous renverra 0). Si la liste contient plusieurs fois la même valeur, c'est l'index de la première valeur rencontrée qui sera renvoyé.

Enfin, pour terminer notre découverte de ce module, voyons comment intercepter des caractères au clavier. Nous avons vu la fonction **getch**... Il suffit donc d'effectuer une boucle sur cette instruction pour récupérer toutes les saisies au clavier et les traiter :

In []:
```
c = None            # getch renvoie le code ASCII associé au caractère.
while c != 27 :     # 27 correspond à la touche [Echap]
    c = win.getch()
    if c == curses.KEY_UP :
        win.addstr(10, 1, "Flèche vers le haut")
    elif c == curses.KEY_DOWN :
        win.addstr(10, 1, "Flèche vers le bas")
    elif c == curses.KEY_LEFT :
        win.addstr(10, 1, "Flèche vers la gauche")
```

```
    elif c == curses.KEY_RIGHT :
        win.addstr(10, 1, "Flèche vers la droite")
```

À chaque touche on associe un code particulier, Ainsi, à la lettre A correspond le code 65, à B correspond le code 66, etc. Ce code est appelé **code ASCII** et permet d'identifier toutes les touches du clavier, y compris les touches spéciales. C'est pourquoi nous pouvons tester en condition de fin de boucle si le code de la touche est 27 : en appuyant sur la touche `[Echap]` on sort de la boucle. Pour les autres touches spéciales, nous utiliserons les raccourcis fournis par le module **curses**. Pour une liste complète des raccourcis, vous pourrez consulter la page http://docs.python.org/3/library/curses.html en section 16.11.3 intitulée "Constants".

6.2 Le module curses dans notre jeu

Pour utiliser **curses** dans notre jeu, nous allons pouvoir récupérer les fonctions que nous avons créées pour tester le module : *init_curses*, *clos_curses* et *color*. Comme nous avons segmenté notre code, seule les fonctions d'affichage devront être modifiées. La plus touchée sera la fonction *affiche_labyrinthe* :

In []:
```
def affiche_labyrinthe(lab, perso, pos_perso, tresor, win, coul):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage [ligne, colonne]
        tresor : caractère représentant le trésor
        win : fenêtre du mode graphique
        coul : liste de couleurs pour le mode graphique

        Pas de valeur de retour
    """
    n_ligne = 0
    for ligne in lab:
        for i in range(1, 4) :
            ligne = ligne.replace(str(i), tresor)
        if  n_ligne == pos_perso[1]:
            win.addstr(n_ligne +1, 10, ligne[0:pos_perso[0]] + perso + \
                       ligne[pos_perso[0] +1:])
            # Coloration du personnage
            win.addstr(n_ligne +1, 10 +pos_perso[0], perso, color("RED", coul))
        else :
            win.addstr(n_ligne +1, 10, ligne)
        n_ligne += 1
```

Pour ne pas toucher à la structure de notre programme, nous continuons à afficher tous les caractères de toutes les lignes et il faut donc afficher une deuxième fois le personnage pour le colorer en rouge. En s'autorisant plus de modifications, comme le module **curses** permet d'afficher des caractères à des coordonnées précises dans la fenêtre, il suffirait d'afficher le labyrinthe une seule fois (et non à chaque fois que l'utilisateur appuie sur une touche) et il faudrait alors seulement effacer le personnage de son ancien emplacement et l'afficher à sa nouvelle position.

Cela peut constituer un bon exercice : partez du programme complet de cette séance et modifiez-le de manière à n'afficher les murs du labyrinthe qu'une seule fois et à colorer les ennemis en vert.

Pour récapituler

- Le module **curses** permet d'améliorer les interfaces en mode console ;

- Lorsque l'on intercepte une exception avec un bloc **try/except**, l'ajout d'un bloc **finally** permet de s'assurer qu'un bloc de code soit exécuté, qu'il y ait ou non une erreur.

Résumé code :

Comme le fichier `Lab.py` était bien structuré, une fois que les fonctions spécifiques au mode "graphique" console ont été ajoutées, peu de lignes sont modifiées :

In []:
```
import os
import random, curses

def init_curses(lignes, cols, pos) :
    """
        Initialisation des paramètres graphiques

        lignes : nombre de lignes (en caractères)
        cols   : nombre de colonnes (en caractères)
        pos    : tuple contenant la position du coin supérieur gauche
                 de la fenêtre graphique

        Valeur de retour :
            La fenêtre curses ayant été créée
    """
    curses.initscr()
    curses.noecho()
    curses.cbreak()
    curses.curs_set(0)

    window = curses.newwin(lignes, cols, pos[0], pos[1])
    window.border(0)
    window.keypad(1)
    return window


def close_curses() :
    """
        Restauration des paramètres graphiques
    """
    # Appel des fonctions inverses que celles invoquées dans init_curses
    curses.echo()
    curses.nocbreak()
    curses.curs_set(1)
    curses.endwin()

def init_colors() :
    """
        Initialisation des couleurs

        Valeur de retour :
            liste contenant le nom des couleurs (index = code couleur)
    """
    curses.start_color()
    curses.init_pair(1, curses.COLOR_RED, curses.COLOR_BLACK)
    curses.init_pair(2, curses.COLOR_GREEN, curses.COLOR_BLACK)
    curses.init_pair(3, curses.COLOR_BLACK, curses.COLOR_BLUE)
    return ["RED", "GREEN", "BLUE"]


def color(code, l_color):
    """
        Sélectionne une couleur
```

```
        code : nom de la couleur
        l_color : liste des couleurs

        Valeur de retour :
            code de couleur curses
    """
    return curses.color_pair(l_color.index(code) +1)


def charge_labyrinthe(nom) :
    """
        Charge le labyrinthe depuis le fichier nom.txt

        nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)

        Valeur de retour :
            - une liste avec les données du labyrinthe
    """
    try :
        fic = open(nom + ".txt", "r")
        data = fic.readlines()
        fic.close()
    except IOError :
        print("Impossible de lire le fichier {}.txt".format(nom))
        os._exit(1)

    for i in range(len(data)) :
        data[i] = data[i].strip()

    return data


def barre_score(data, win, coul) :
    """
        Barre de score affichant les données du jeu

        data : dictionnaire de données de la barre de score
        win  : fenêtre graphique
        cou  : liste de couleurs pour le mode graphique

        Pas de valeur de retour
    """
    barre = "PV : {:2d}      PO : {:4d}      Level : {:3d}"
    win.addstr(21, 1, barre.format(data["pv"],
                                   data["po"], data["level"]), color("BLUE", coul))


def affiche_labyrinthe(lab, perso, pos_perso, tresor, win, coul):
    """
        Affichage d'un labyrinthe

        lab : Variable contenant le labyrinthe
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage [ligne, colonne]
        tresor : caractère représentant le trésor
        win : fenêtre du mode graphique
        coul : liste de couleurs pour le mode graphique

        Pas de valeur de retour
    """
    n_ligne = 0
    for ligne in lab:
        for i in range(1, 4) :
            ligne = ligne.replace(str(i), tresor)
        if  n_ligne == pos_perso[1]:
            win.addstr(n_ligne +1, 10, ligne[0:pos_perso[0]] + perso + \
                       ligne[pos_perso[0] +1:])
```

```
            # Coloration du personnage
            win.addstr(n_ligne +1, 10 +pos_perso[0], perso, color("RED", coul))
        else :
            win.addstr(n_ligne +1, 10, ligne)
        n_ligne += 1


def decouverte_tresor(categorie, data) :
    """
        Incrémente le nombre de pièces d'or du joueur en fonction du trésor

        categorie : type de trésor
        - 1 : entre 1 et 5 po
        - 2 : entre 5 et 10 po
        - 3 : entre 0 et 25 po
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    if categorie == "1" :
        data["po"] = data["po"] + random.randint(1, 5)
    elif categorie == "2" :
        data["po"] = data["po"] + random.randint(5, 10)
    else :
        data["po"] = data["po"] + random.randint(0, 25)


def combat(data) :
    """
        Détermine le nombre de points de vie perdus lors d'un combat

        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
    """
    de = random.randint(1, 10)    # Tirage du dé entre 1 et 10
    if de == 1 :
        data["pv"] = data["pv"] - random.randint(5, 10)
    elif de >= 2 and de <= 4 :
        data["pv"] = data["pv"] - random.randint(1, 5)


def verification_deplacement(lab, pos_col, pos_ligne, data):
    """
        Indique si le déplacement du personnage est autorisé ou pas.

        lab : Labyrinthe
        pos_ligne : position du personnage sur les lignes
        pos_col : position du personnage sur les colonnes
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)

        Valeurs de retour :
          None : déplacement interdit
          [col, ligne] : déplacement autorisé sur la case indiquée par la liste
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    # Teste si le déplacement conduit le personnage en dehors de l'aire de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
       pos_col > (n_cols -1) :
        return None
    elif lab[pos_ligne][pos_col] == "O" :
        # Une position hors labyrinthe indique la victoire
        return [-1, -1]
    elif lab[pos_ligne][pos_col] == "1" or lab[pos_ligne][pos_col] == "2" or \
         lab[pos_ligne][pos_col] == "3" :
        # teste si le personnage se déplace sur un trésor
        # Découverte d'un trésor
        # fonction qui calcule le montant du butin
```

```
            decouverte_tresor(lab[pos_ligne][pos_col], data)
            # On supprime le trésor découvert
            lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
                             lab[pos_ligne][pos_col +1:]
            return [pos_col, pos_ligne]
        elif lab[pos_ligne][pos_col] == "$" :
            # Rencontre d'un ennemi
            combat(data)
            lab[pos_ligne] = lab[pos_ligne][:pos_col] + " " + \
                             lab[pos_ligne][pos_col +1:]
            return [pos_col, pos_ligne]
        elif lab[pos_ligne][pos_col] != " " :
            return None
        else :
            return [pos_col, pos_ligne]


def choix_joueur(lab, pos_perso, data, win):
    """
        Demande au joueur de saisir son déplacement et vérifie s'il est possible.
        Si ce n'est pas le cas affiche un message, sinon modifie la position
        du perso dans la liste pos_perso

        lab : Labyrinthe
        pos_perso : liste contenant la position du personnage [colonne, ligne]
        data : données de jeu (niveaux, nombre de pièces d'or et points de vie)
        win : fenêtre graphique

        Pas de valeur de retour
    """
    dep = None
    choix = win.getch()
    if choix == curses.KEY_UP :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] -1, data)
    elif choix == curses.KEY_DOWN :
        dep = verification_deplacement(lab, pos_perso[0], pos_perso[1] +1, data)
    elif choix == curses.KEY_LEFT :
        dep = verification_deplacement(lab, pos_perso[0] -1, pos_perso[1], data)
    elif choix == curses.KEY_RIGHT :
        dep = verification_deplacement(lab, pos_perso[0] +1, pos_perso[1], data)
    elif choix == 27 :
        close_curses()
        os._exit(1)
    if dep != None :
        pos_perso[0] = dep[0]
        pos_perso[1] = dep[1]




def jeu(level, data, perso, pos_perso, tresor, win, coul):
    """
        Boucle principale du jeu. Affiche le labyrinthe dans ses différents
        états après les déplacements du joueur.

        level : Labyrinthe
        data : dictionnaire contenant
               - level : le numéro de niveau
               - po    : le nombre de pièces d'or
               - pv    : le nombre de points de vie
        perso : caractère représentant le personnage
        pos_perso : liste contenant la position du personnage [colonne, ligne]
        tresor : caractère représentant le trésor
        win  : fenêtre graphique
```

```
        coul : liste de couleurs pour la fenêtre graphique
    """
    while True :
        affiche_labyrinthe(level, perso, pos_perso, tresor, win, coul)
        barre_score(data, win, coul)
        if data["pv"] <= 0 :
            win.addstr(1, 20, "Vous avez PERDU...", color("RED", coul))
            win.getch()
            close_curses()
            os._exit(1)
        choix_joueur(level, pos_perso, data, win)
        if pos_perso == [-1, -1] :
            win.addstr(22, 1, "Vous avez passé le niveau !", color("RED", coul))
            win.addstr(23, 1, "Appuyez sur une touche pour continuez",
                                                    color("RED", coul))
            win.getch()
            win.addstr(1, 20, " "*50)
            win.addstr(1, 21, " "*50)
            break
```

Même constat pour le fichier `Jeu_lab.py`, qui délègue l'essentiel du travail au module `Lab`.

In []:
```
#!/usr/bin/python3

import Lab

if __name__ == "__main__" :
    # Initialisation du personnage
    perso           = "X"
    pos_perso       = [1,1]
    tresor          = "#"
    n_levels_total = 20
    data = {
            "po" : 0,
            "pv" : 25,
            "level" : None
            }

    # Initialisation de l'affichage graphique
    win = Lab.init_curses(25, 41, (0,0))
    # Initialisation des couleurs
    coul = Lab.init_colors()

    # Lancement de la partie
    for n_level in range(1, n_levels_total +1) :
        level = Lab.charge_labyrinthe("level_" + str(n_level))
        data["level"] = n_level
        Lab.jeu(level, data, perso, pos_perso, tresor, win, coul)
    win.addstr(1, 22, "Vous avez gagné !!", Lab.color("RED", coul))
    win.getch()
    Lab.close_curses()
```

#

7 Séance : Passage en mode graphique

RetourNotre jeu es terminé, mais il s'exécute dans une console et il n'est pas vraiment joli. Nous allons donc créer pour lui une interface graphique... Ce qui va complètement modifier notre code !

- Une fenêtre graphique
- Des objets
- Dessiner dans une fenêtre
- Gestion des événements

Il existe de nombreux modules Python permettant de créer des interfaces graphiques. Certains sont même spécialement conçus pour le développement de jeux. Nous utiliserons ici un environnement graphique très simple et installé par défaut sur tous les systèmes : **Tk**.

Le problème avec le module **tkinter** que nous allons utiliser, c'est qu'il n'est pas écrit de la même manière que notre code, une architecture particulière a été utilisée : la programmation orientée objet. Ce style de programmation est un peu plus complexe que ce que nous avons fait jusqu'alors, mais il possède une grande qualité : le code est bien structuré et facilement réutilisable. Nous ne pourrons pas aborder en une séance tous les principes de la programmation orientée objet. Nous profiterons simplement du fait d'utiliser **tkinter** pour découvrir certaines notions clés.

7.1 Une fenêtre graphique

L'ouverture d'une fenêtre graphique se fait en quelques lignes depuis le shell Python :

In []:
```
>>> from tkinter import *
>>> fenetre = TK()
```

Dans un script indépendant, une dernière ligne est nécessaire pour lancer l'affichage de la fenêtre graphique :

In []:
```
fenetre.mainloop()
```

Il s'agit de ce que l'on appelle la boucle événementielle : une boucle infinie qui s'exécute en attendant de récupérer les événements de l'utilisateur, tels que l'appui sur une touche du clavier, le déplacement de la souris, etc.

Nous avons chargé le module **tkinter** à l'aide de la syntaxe **from module import *** et non **import module** : c'est une autre façon de charger un module qu'il vaut mieux réserver aux modules qui fournissent des objets. En effet, avec cette écriture, vous n'avez plus besoin de nommer le module devant le nom d'une fonction. Vous pensez que c'est plus pratique ? que se passe-t-il si deux modules `A` et `B` contiennent chacun une fonction *fct* qui ne réalise pas la même tâche ?

In [3]:
```
#Fichier A.py
def fct() :
    print("Je suis dans A !")
```

In [4]:
```
#Fichier B.py
def fct() :
    print("Je suis dans B !")
```

In []:
```
>>> from A import *
>>> from B import *
>>> fct()
Je suis dans B !
```

La première version de la fonction, présente dans le module `A`, a donc été écrasée par la deuxième version et nous n'y avons plus accès. En utilisant la syntaxe `import` ce n'est pas le cas :

In []:
```
>>> import A
>>> import B
>>> A.fct()
Je suis dans A !
>>> B.fct()
Je suis dans B !
```

Comme les objets sont définis dans un bloc qui leur est propre et porte leur nom, ce risque d'erreur n'est pas possible. Mais au fait, qu'est-ce qu'un objet ?

7.2 Des objets

Dans la vie, nous sommes entourés d'objets : un livre est un objet, une page du livre est un objet... Bref, tout est objet ! Et comme vous l'aurez remarqué, un livre étant composé de pages, des objets peuvent être construits en utilisant d'autres objets.

En informatique, un objet est une entité représentant un élément à partir de ces caractéristiques fondamentales. Un livre a forcément un titre, un ou des auteur(s) et un éditeur. Ces informations seront stockées dans des variables attachées à l'objet livre : pour un livre donné nous aurons un titre, etc. Ces variables portent un nom spécifique pour bien montrer qu'elles définissent un objet. On les appelle **des attributs** de l'objet.

Il n'existe pas "un" livre, mais "des" livres. Lorsque nous définissons un objet, nous ne parlons pas d'un objet particulier, mais de l'ensemble des objets qu'il représente. On parle alors d'**une classe**. La classe "livre" permet de définir comment un livre doit être fabriqué et, plus tard, on utilisera ce modèle pour créer des livres qui seront alors **des instances** de la classe "livre" (des livres réels, qui ne sont pas seulement des définitions).

Prenons l'exemple des Lego. Une boîte de Lego contient un plan et des briques : le plan seul, ou une brique seule n'ont pas grand intérêt. Par contre, si l'on se sert du plan (la classe) pour assembler les briques, on peut créer un objet qui sera une instance du plan de départ. Si la boîte correspond à une voiture, nous obtiendrons une voiture et si nous avons d'autres briques, de couleurs différentes, en suivant le plan nous obtiendrons un autre modèle de voiture.

D'un point de vue informatique, des fonctions vont pouvoir être liées à un objet. Ces fonctions particulières, appelées **méthodes**, connaissent toutes la structure de l'objet et ont un accès direct à ses attributs. Voyons un exemple en Python, avec la classe `Livre` que nous stockerons dans un fichier du même nom `livre.py` :

In []:
```
class Livre(object) : # Définition de la façon de construire un objet "Livre"

    def __init__(self, auteur, titre, editeur) :
        self.auteur  = auteur
        self.titre   = titre
        self.editeur = editeur
        # Définition de la méthode permettant de construire l'objet.
        # Trois attributs définissent ce qu'est un livre.

    def couverture(self) :  # Méthode affichant la couverture d'un livre
        print(self.titre)
        print("de", self.auteur)
        print("édité par", self.editeur)
```

Le mot-clé **self**, présent dans de nombreuses lignes, désigne l'objet courant et permet de faire la distinction entre les variables et les attributs. Toutes les méthodes prennent pour premier paramètre l'objet courant. La méthode **__init__** est une méthode particulière appelée **constructeur**. C'est elle qui sera appelée quand nous créerons des instances de la classe :

In []:
```
>>> from Livre import *
>>> l1 = Livre("David Gemmel", "Druss la légende", "Milady")
>>> l2 = Livre("Richard Castle", "Vague de chaleur", "Gina Cowell")
>>> l1.couverture()
Druss la légende
de David Gemmel
édité par Milady
>>> l2.couverture()
Vague de chaleur
de Richard Castle
édité par Gina Cowell
```

`l1` et `l2` sont des instances de la classe `Livre`. Ce sont donc des objets Livre créés grâce au constructeur `__init__`, qui est appelé indirectement dans les lignes `l1 = Livre(...)` et `l2 = Livre(...)` (pour vous en convaincre, vous pouvez ajoutez une instruction `print` dans le constructeur). On peut lire le continu de la couverture d'un livre grâce à la méthode `couverture`. Vous noterez que l'on ne transmet aucun paramètre à `couverture`... C'est l'opérateur . qui fait le lien et *transforme* l'instance `l1` ou `l2` en `self` lorsque l'on se retrouve dans la classe. Nous avons déjà utilisé de nombreuses méthodes sans les nommer, car en Python, tout ce que l'on manipule est un objet (entier, chaîne de caractères, etc.)

Pour revenir à l'interface graphique, le code `fenetre = Tk()` crée une instance de la classe `Tk`, qui est une fenêtre graphique. Nous allons maintenant pouvoir ajouter des éléments graphiques en comprenant ce que nous faisons.

7.3 Dessiner dans une fenêtre

Pour notre jeu, nous avons simplement besoin d'afficher de petites images (*sprites*) pour créer le labyrinthe et représenter les ennemis, les trésors, la sortie et le joueur. Il faudra 5 sprites différents, que l'on peut créer à l'aide de GIMP, ou en se baszant sur des images existantes (attention à la licence !). Les images que j'utilise ici ont une taille de 30×30 px.

Regardons dans un premier temps comment afficher un seul sprite :

In []:
```
from tkinter import *

fenetre = Tk()
fenetre.title("Jeu Labyrinthe")
# Ajout d'un titre en haut de la fenêtre graphique
#Création d'un élément graphique permettant de dessiner à l'intérieur
can = Canvas(fenetre, width = 500, height = 500)

photo_wall = PhotoImage(file = "sprites/wall.gif")

#Chargement de l'image wall.gif et affichage en position (0,0)
sprite_wall = can.create_image(0, 0, anchor = NW, image = photo_wall)

can.pack()  # Positionnement effectif du canevas dans la fenêtre principale

fenetre.mainloop() # Boucle événementielle
```

Lors de la création d'une instance de la classe **Canvas**, nous donnons en premier paramètre la fenêtre : il s'agit de l'objet dans lequel nous allons placer le canevas. La construction d'une interface graphique se fait sur la base d'empilements successifs : un bouton dans une fenêtre, une image dans le bouton, etc. Une fois la liaison faite entre le contenant et le contenu, il faut en quelque sorte la valider, la rendre effective. C'est ce que fait la méthode `pack`.

Pour afficher le labyrinthe, nous utiliserons la fonction de lecture que nous avions définie et en parcourant les caractères de la liste obtenue, nous pourrons déterminer quel sprite afficher :

In []:
```
def affiche_labyrinthe(lab, fenetre, size_sprite, pos_perso) :
    """
        Affichage d'un labyrinthe.

        lab : Variable contenant le labyrinthe
        fenetre: Fenêtre graphique
        size_sprite : Taille des sprites en pixels
        pos_perso : Liste contenant la position du personnage

        Valeur de retour :
            Tuple contenant le canevas, le sprite du personnage et un
            dictionnaire des images utilisées pour les sprites
    """
    can = Canvas(fenetre, width = 600, height = 600)

    photo_wall     = PhotoImage(file = "sprites/wall.gif")
    photo_treasure = PhotoImage(file = "sprites/treasure.gif")
    photo_ennemy   = PhotoImage(file = "sprites/ennemy.gif")
    photo_exit     = PhotoImage(file = "sprites/exit.gif")
    photo_hero     = PhotoImage(file = "sprites/hero.gif")

    n_ligne = 0
    for ligne in lab :
        n_col = 0
        for car in ligne :
            # Murs
            if car == "+" or car == "-" or car == "|" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_wall)
            # Trésors
            if car == "1" or car == "2" or car == "3" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_treasure)
            # Ennemis
            if car == "$" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_ennemy)
            # Sortie
            if car == "O" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_exit)
            n_col += 1
        n_ligne += 1

    # Affichage du personnage
    sprite_hero = can.create_image(pos_perso[0] + pos_perso[0] * size_sprite,
                                   pos_perso[1] + pos_perso[1] * size_sprite,
                                   anchor = NW,image = photo_hero)

    can.pack()

    return (can, sprite_hero, {
        "hero"    : photo_hero,
        "wall"    : photo_wall,
        "treasure": photo_treasure,
        "ennemy"  : photo_ennemy,
        "exit"    : photo_exit})
```

Ce code est une réécriture de notre fonction *affiche_labyrinthe* avec les sprites : la structure reste pratiquement in-

changée. Notez toutefois qu'il y a ici beaucoup de valeurs qui sont retournées par la fonction : nous devons garder accès au canevas pour le modifier, au sprite du personnage pour le déplacer et aux différentes photos pour les afficher (si elles ne sont plus en mémoire, rien n'apparaîtra à l'écran).

Pour appeler cette fonction, après avoir créé la fenêtre graphique, il faut charger un niveau et penser à récupérer les valeurs de retour (le code complet est disponible en fin de séance) :

In []:
```
level = Lab.charge_labyrinthe("level_1")

(canvas, sprite_perso, photos) = Lab.affiche_labyrinthe(level, fenetre,
                                                        size_sprite, pos_perso)
```

Passons maintenant à la saisie des déplacements du personnage.

7.4 Gestion des événements

En mode graphique, toutes les actions de l'utilisateur sont codées sous forme d'événements. Un clic de souris, un appui sur une touche sont des événements. Chaque élément graphique de l'interface peut être associé à un ou plusieurs événement(s). Par exemple, si nous voulons afficher un message lorsque l'utilisateur clique dans la fenêtre (sur le canevas), nous ferons :

In []:
```
canvas.bind("<Button-1>", click)
```

Nous "lions" l'événement `<Button-1>`, c'est-à-dire un clic gauche, sur l'objet graphique **canvas**. Lorsque l'événement se produit, la fonction *click* sera exécutée. Cette fonction doit être écrite de la manière suivante :

In []:
```
def click(event) :
    print("Vu ! Vous avez cliqué sur le jeu !")
```

Comme vous pouvez le constater, elle possède un paramètre... Alors que nous n'en avons transmis aucun. C'est normal, le système transmet automatiquement un paramètre d'informations sur le type d'événement intercepté. Ici, lors du clic de l'utilisateur, une variable est créée et transmise à la fonction *click* dont le nom était passé en paramètre à la méthode `bind`. Il s'agit bien du nom de la fonction et non d'un appel, comme vous pouvez le vérifier dans le shell Python avec un test sur la fonction `print` :

In []:
```
>>> print("Hello")
Hello
>>> print
<built-in function print>
```

La deuxième syntaxe est une référence au code de la fonction `print`. Tant qu'il n'y a pas de parenthèses, l'appel n'est pas exécuté. On pourrait même stocker cette référence dans une variable pour y faire appel plus tard :

In []:
```
>>> ecrire = print
>>> ecrire("Hello")
Hello
```

La variable *ecrire* est une référence à la fonction `print`. C'est ce que l'on appel **un pointeur** : la variable pointe vers une zone de la mémoire qui contient un certain code (ici une fonction).

Le problème de ce mécanisme, lorsqu'on l'utilise pour intercepter des événements, est que l'on ne peut pas transmettre de paramètres à la fonction puisqu'on ne doit donner que sa référence et non son appel direct...

Pour contourner ce problème, nous allons utiliser des fonctions anonymes ou lambda fonctions. Le principe consiste à créer une référence à une fonction construite par appel à une fonction contenant des paramètres.

Exemple :

In []:
```
>>> f = lambda txt = "Hello" : print(txt)
>>> f
<function <lambda> at 0x7f06a48b3d10>
>>> f()
Hello
```

`f` est une référence vers une fonction créée en appelant la fonction `print` avec le paramètre `txt`. Lorsque nous exécutons `f` à l'aide des parenthèses, nous obtenons l'affichage du contenu de la variable `txt`.

Concrètement, dans la fonction *init_touches*, qui va définir les actions à effectuer en fonction des touches sur lesquelles l'utilisateur va appuyer, nous allons utiliser des fonctions anonymes pour déplacer le personnage et fermer la fenêtre graphique :

In []:
```
def init_touches(fenetre, canvas, lab, pos_perso, perso) :
    """
        Initialisation du comportement des touches du clavier

        canvas    : canevas où afficher les sprites
        lab       : liste contenant le labyrinthe
        pos_perso : position courante du personnage
        perso     : sprite représentant le personnage

        Pas de valeur de retour
    """
    fenetre.bind("<Right>", lambda event, can = canvas, l = lab,
                 pos = pos_perso,
                 p = perso : deplacement(event, can, "right", l, pos, p))
    # Appels des fonctions anonymes
    # Autres cas
    # ...
    fenetre.bind("<Escape>", lambda event, fen = fenetre : destroy(event, fen))
```

Les fonctions *deplacement* et *destroy* vont être appelées lors de l'appui sur la flèche droite ou la touche [Echap]. Le premier paramètre est l'objet événement transmis automatiquement à la fonction.

In []:
```
def deplacement(event, can, dep, lab, pos_perso, perso):
    """
        Déplacement du personnage

        event   : objet décrivant l'événement ayant déclenché
                              l'appel à cette fonction
        can     : canevas où afficher les sprites
        dep     : type de déplacement ("up", "down", "left" ou "right")
        lab     : liste contenant le labyrinthe
        pos_perso : position courante du personnage
        perso   : sprite représentant le personnage

        Pas de valeur de retour
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    pos_col, pos_ligne = [pos_perso[0], pos_perso[1]]

    # Déplacement vers la droite
    if dep == "right" :
        pos_col += 1

    # Teste si le déplacement conduit le personnage en dehors de l'aire de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
       pos_col > (n_cols -1) :
        return None
```

```
    # Si le déplacement est possible sur une case vide
    if lab[pos_ligne][pos_col] == " " :
        can.coords(perso, pos_col + pos_col * 30, pos_ligne + pos_ligne * 30)
        # modif de la liste pos_perso pour que les nouvelles valeurs
        # soient accessibles depuis les autres fonctions
        del pos_perso[0]
        del pos_perso[0]
        pos_perso.append(pos_col)
        pos.perso.append(pos_ligne)

def destroy(event, fenetre) :
    """
        Fermeture de la fenêtre graphique

        event   : objet décrivant l'événement ayant déclenché l'appel à cette
                  fonction
        fenetre : fenêtre graphique

        Pas de valeur de retour
    """
    fenetre.destroy()   # Appel de la méthode destroy de la fenêtre graphique
#( à ne pas confondre avec la fonction dans laquelle on se trouve)
```

Ici, il a fallu modifier la structure de la fonction chargée du déplacement du personnage : les fonctions sont appelées suite à des événements liés à des fonctions et nous ne pouvons pas récupérer de valeurs en retour. Nous modifions donc directement la liste `pos_perso`, de manière à ce que les modifications soient accessibles depuis n'importe quelle fonction, qu'elles restent persistantes.

L'instruction `del` permet de supprimer des éléments d'une liste (en les décalant). Donc si, sur une liste de deux éléments, on supprime le premier élément, il ne reste plus qu'une liste d'un seul élément et, pour qu'elle soit vide, il faut de nouveau supprimer le premier élément.

L'appel de la fonction *init_touches* se fait dans le programme principal en lui passant en parmètre la fenêtre graphique, le canevas, la liste contenant le labyrinthe, la position et le sprite du personnage :

In []:
```
Lab.init_touches(fenetre, canvas, level, pos_perso, sprite_perso)
```

7.5 Conclusion

Je n'ai volontairement pas codé en mode graphique le jeu tel que nous l'avions terminé en séance 6. À partir de tous les éléments que nous avons pu voir lors de ces séances, vous devriez être capable d'écrire le code complet de notre jeu en mode graphique. N'oubliez pas de récupérer les variables associées aux sprites qui peuvent être déplacés (ou enlevés) et d'afficher la barre des scores en utilisant la méthode `create_text` qui, comme `create_image`, prend en premier paramètre la position du texte suivie du texte à afficher.

N'hésitez pas à utiliser l'aide en ligne ou les ressources du site python.org pour vous aider et surtout ne vous découragez pas : il faut environ trois mois à un informaticien pour apprendre un nouveau langage. Seules la pratique et la correction des erreurs vous permettront de comprendre réellement ce que vous faites.

Pour récapituler

- Un **objet** est une structure permettant une programmation modulaire et réutilisable. La définition d'un objet se fait dans une classe ;
- Un **attribut** est une variable attachée à un objet et permettant de le définir ;
- Une **méthode** est une fonction attachée à un objet et qui a accès à ses attributs ;
- Une **instance** est une variable créée en se servant d'une classe pour modèle ;
- **del** est une instruction permettant de supprimer des éléments dans une liste.

Résumé code :

Le fichier `Lab.py` contient les fonctions principales du jeu :

In []:
```
import os
import random
from tkinter import *

def charge_labyrinthe(nom) :
    """
        Charge le labyrinthe depuis le fichier nom.txt

        nom : nom du fichier contenant le labyrinthe (sans l'extension .txt)

        Valeur de retour :
            - une liste avec les données du labyrinthe
    """
    try :
        fic = open(nom + ".txt", "r")
        data = fic.readlines()
        fic.close()
    except IOError :
        print("Impossible de lire le fichier {}.txt".format(nom))
        os._exit(1)

    for i in range(len(data)) :
        data[i] = data[i].strip()

    return data


def affiche_labyrinthe(lab, fenetre, size_sprite, pos_perso):
    """
        Affichage d'un labyrinthe.

        lab : Variable contenant le labyrinthe
        fenetre: Fenêtre graphique
        size_sprite : Taille des sprites en pixels
        pos_perso : Liste contenant la position du personnage

        Valeur de retour :
            Tuple contenant le canevas, le sprite du personnage et un
            dictionnaire des images utilisées pour les sprites
    """
    can = Canvas(fenetre, width = 600, height = 600)

    photo_wall     = PhotoImage(file = "sprites/wall.gif")
    photo_treasure = PhotoImage(file = "sprites/treasure.gif")
```

```
    photo_ennemy   = PhotoImage(file = "sprites/ennemy.gif")
    photo_exit     = PhotoImage(file = "sprites/exit.gif")
    photo_hero     = PhotoImage(file = "sprites/hero.gif")

    n_ligne = 0
    for ligne in lab :
        n_col = 0
        for car in ligne :
            # Murs
            if car == "+" or car == "-" or car == "|" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_wall)
            # Trésors
            if car == "1" or car == "2" or car == "3" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_treasure)
            # Ennemis
            if car == "$" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_ennemy)
            # Sortie
            if car == "O" :
                can.create_image(n_col + n_col * size_sprite,
                                 n_ligne + n_ligne * size_sprite, anchor = NW,
                                 image = photo_exit)
            n_col += 1
        n_ligne += 1

    # Affichage du personnage
    sprite_hero = can.create_image(pos_perso[0] + pos_perso[0] * size_sprite,
                                   pos_perso[1] + pos_perso[1] * size_sprite,
                                                anchor = NW,image = photo_hero)

    can.pack()

    return (can, sprite_hero, {
        "hero"    : photo_hero,
        "wall"    : photo_wall,
        "treasure": photo_treasure,
        "ennemy"  : photo_ennemy,
        "exit"    : photo_exit})


def deplacement(event, can, dep, lab, pos_perso, perso):
    """
        Déplacement du personnage

        event   : objet décrivant l'événement ayant déclenché l'appel à cette
                  fonction
        can     : canevas où afficher les sprites
        dep     : type de déplacement ("up", "down", "left" ou "right")
        lab     : liste contenant le labyrinthe
        pos_perso : position courante du personnage
        perso   : sprite représentant le personnage

        Pas de valeur de retour
    """
    # Calcul de la taille du labyrinthe
    n_cols = len(lab[0])
    n_lignes = len(lab)
    pos_col, pos_ligne = [pos_perso[0], pos_perso[1]]

    # Déplacement vers la droite
    if dep == "right" :
```

```
        pos_col += 1

    # Teste si le déplacement conduit le personnage en dehors de l'aire de jeu
    if pos_ligne < 0 or pos_col < 0 or pos_ligne > (n_lignes -1) or \
       pos_col > (n_cols -1) :
        return None

    # Si le déplacement est possible sur une case vide
    if lab[pos_ligne][pos_col] == " " :
        can.coords(perso, pos_col + pos_col * 30, pos_ligne + pos_ligne * 30)

        del pos_perso[0]
        del pos_perso[0]
        pos_perso.append(pos_col)
        pos.perso.append(pos_ligne)


def destroy(event, fenetre) :
    """
        Fermeture de la fenêtre graphique

        event   : objet décrivant l'événement ayant déclenché l'appel à cette
                  fonction
        fenetre : fenêtre graphique

        Pas de valeur de retour
    """
    fenetre.destroy()


def init_touches(fenetre, canvas, lab, pos_perso, perso) :
    """
        Initialisation du comportement des touches du clavier

        canvas    : canevas où afficher les sprites
        lab       : liste contenant le labyrinthe
        pos_perso : position courante du personnage
        perso     : sprite représentant le personnage

        Pas de valeur de retour
    """
    fenetre.bind("<Right>", lambda event, can = canvas, l = lab,
                 pos = pos_perso,
                 p = perso: deplacement(event, can, "right", l, pos, p))
    # Autres cas
    # ...
    fenetre.bind("<Escape>", lambda event, fen = fenetre : destroy(event, fen))
```

Le fichier `Jeu_lab.py` est le fichier principal qui va utiliser le module `Lab` pour lancer le jeu :

In []:
```
#!/usr/bin/python3

import Lab
from tkinter import *

if __name__ == "__main__" :
    # Initialisation du personnage
    perso          = "X"
    pos_perso      = [1,1]
    tresor         = "#"
    n_levels_total = 20
    data = {
            "po" : 0,
            "pv" : 25,
            "level" : 1
            }
    size_sprite = 29
```

```
    # Initialisation de l'affichage graphique
    fenetre = Tk()
    fenetre.title("Jeu Labyrinthe")

    # Lancement de la partie
    level = Lab.charge_labyrinthe("level_1")

    (canvas, sprite_perso, photos) = Lab.affiche_labyrinthe(level, fenetre,
                                     size_sprite, pos_perso)
    Lab.init_touches(fenetre, canvas, level, pos_perso, sprite_perso)

    # Boucle événementielle
    fenetre.mainloop()
```