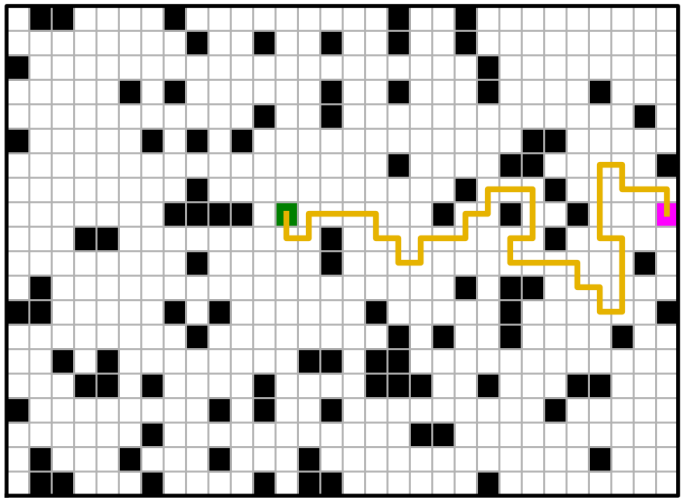
T.C.d'I. — TPC² — TP

GRILLES

Alors voilà : on a une grille, des obstacles, une case de départ (en rose), une case d'arrivée (en vert), et il faut aller de l'une à l'autre.



§1. Grilles

Exercice 1 — Que fait le programme ci-dessous ?

```
def CréerGrille(M, N) :
    return [[None for j in range(N)] for i in range(M)]
```

Exercice 2 — Dans tout le reste de cette fiche, on appellera *grille* tout objet créé par le programme précédent. La *taille* d'une grille sera le couple (M, N). Écrire un programme `Taille(G)` qui renvoie ce couple.

Exercice 3 — Écrire le programme `Copie(G)` qui fait ce qu'on imagine.

§2. Cases aléatoires

Exercice 4 — Écrire un programme `CasesVides(G)` qui construit et renvoie la liste des couples (i, j) tels que `G[`i`][`j`]` contient la valeur `None`.

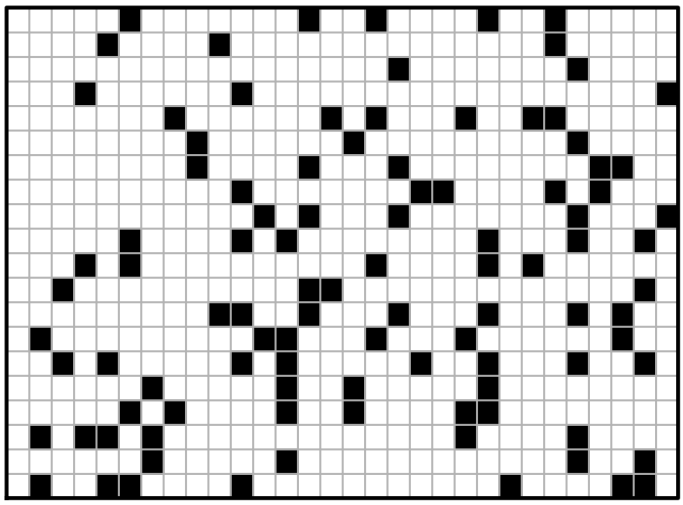
Exercice 5 — On importe les deux fonctions ci-dessous. En utilisant la documentation, préciser ce qu'elles font.

```
from random import randint as EntAléa, shuffle as Mélanger
```

Exercice 6 — Nous allons murer certaines cases d'une grille. Ce qui, pour ce qui nous concerne, va consister à attribuer la valeur `"MUR"` à certaines cases.

1) Expliquer comment, avec la fonction `Mélanger`, choisir aléatoirement n éléments distincts (c'est-à-dire situés dans des cases différentes, même s'ils peuvent avoir la même valeur) d'une liste.

2) En déduire un programme `CréerMurs(G, n)` qui mure n cases vides distinctes de la grille `G`.



§3. Représentation graphique

Dans cette partie, on va programmer les fonctions auxiliaires du grand programme suivant.

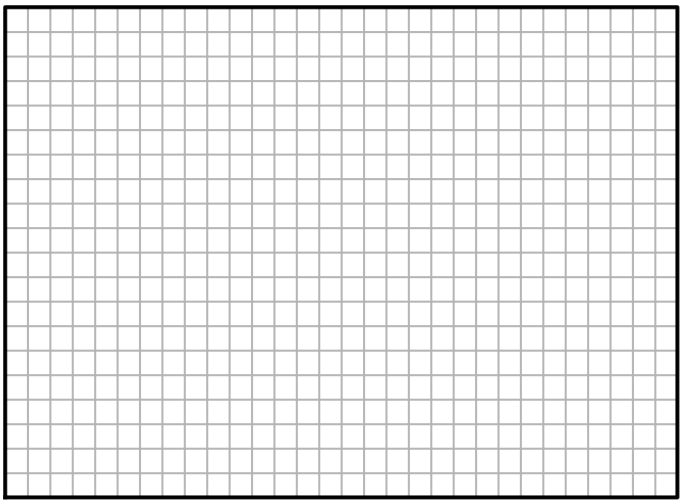
```
from matplotlib.pyplot import axis, fill, plot, show
def DessinerTout(G, CasesSpéciales, Chemins) :
    (M, N) = Taille(G)
    DessinerGrille(M, N)
    DessinerMurs(G)
    for (i, j, couleur) in CasesSpéciales :
        DessinerCase(i, j, couleur)
    for (ch, couleur) in Chemins :
        DessinerChemin(ch, couleur)
    axis([-1, N, -1, M]) ; show()
```

Exercice 7 — La case (i, j) sera représentée par un petit carré, de centre (j, i), et de côté 1.

1) Expliquer pourquoi (j, i) et pas (i, j).

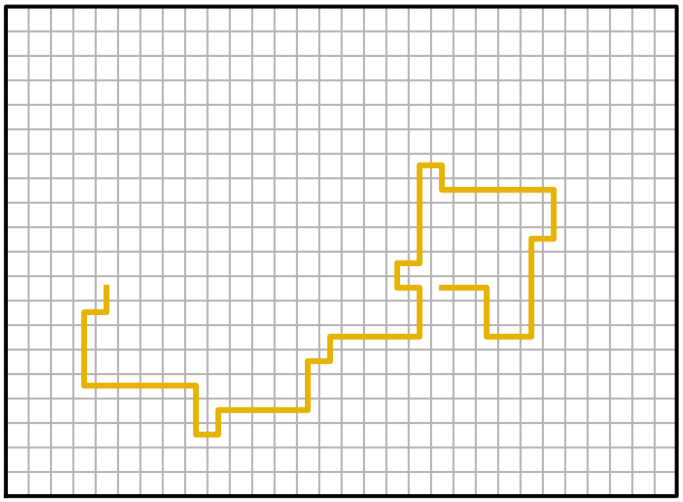
2) En déduire le programme `DessinerCase(i, j, couleur)` qui peint la case (i, j) avec la `couleur`. On utilisera l'instruction `fill`.

Exercice 8 — Écrire un programme `DessinerGrille(M, N)` qui dessine le quadrillage (en gris clair) et les bords (en noir et un peu plus épais).



Exercice 9 — Écrire le programme `DessinerMurs(G)` qui peint en noir les cases murées de la grille.

Exercice 10 — Appelons *chemin* une suite de cases (supposées vides) d'une grille, rangées dans une liste `ch`, telle que pour tout indice k, `ch[`k`]` et `ch[`$k+1$`]` soient des cases adjacentes de la grille. Écrire un programme `DessinerChemin(ch, couleur)` qui fait ce qu'on imagine.

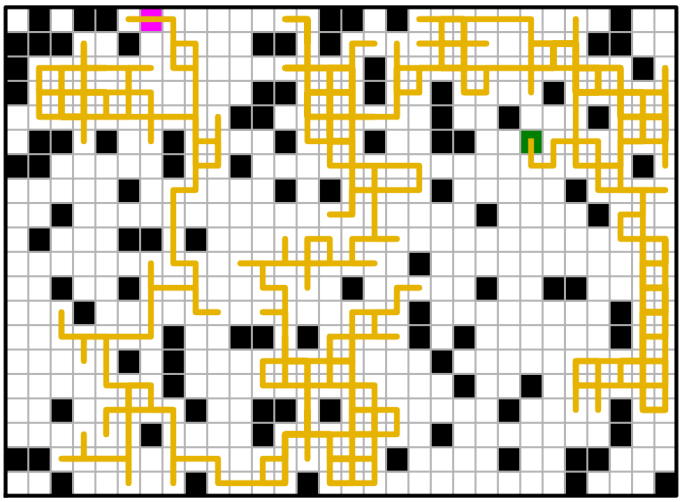


§4. Chemins

Exercice 11 — Écrire un programme `VoisinesVides(G,` i`,` j`)` qui étant donnée une grille `G` et une case (i, j) construit et renvoie la liste des couples (i', j') correspondant à des cases à adjacentes à `G[`i`][`j`]`, et contenant la valeur `None`. Une case a au plus quatre voisines, éventuellement moins si elle est située sur un bord ou dans un coin.

Exercice 12 — Écrire le programme `Choisir(L)` qui renvoie un élément aléatoire d'une liste.

Exercice 13 — Déduire des deux exercices précédents un programme `MarcheAléatoire(G,` i_0`,` j_0`,` i_F`,` j_F`)` qui construit un chemin reliant les cases (i_0, j_0) et (i_F, j_F). On utilisera un alorithme naïf, qui part de la case (i_0, j_0), puis qui se déplace aléatoirement de voisines en voisines, jusqu'à trouver la case cible. Ce chemin passe sans doute de nombreuses fois par les mêmes cases.



Exercice 14

1) Écrire un programme `ChercherRépétition(L)` qui étant donnée une liste renvoie un couple (i, j), avec $i < j$, tel que `L[`i`]` est égal à `L[`j`]`.

2) En déduire un programme `Simplifier(ch)` qui prend en argument un chemin, cherche une case par laquelle il passe deux fois (disons `ch[`k_1`]` et `ch[`k_2`]`), et renvoie le chemin obtenu en supprimant la boucle correspondante (donc la portion `ch[`k_1 `:` k_2`]`). Ce programme renverra `ch` s'il ne comporte aucune boucle.

3) Écrire finalement le programme `SimplifierAuMaximum(ch)` qui construit et renvoie le chemin obtenu en supprimant toutes les boucles de `ch`.

Exercice 15

1) Écrire un programme `TrouverChemin(G,` i_0`,` j_0`,` i_F`,` j_F`)` qui construit et renvoie un chemin sans boucle reliant les cases (i_0, j_0) et (i_F, j_F).

2) Tester tout ce qui précède : en générant une grille vide, en y ajoutant des murs, en choisissant une case de départ et une case d'arrivée aléatoires, et en calculant un chemin les reliant. Et évidemment, on veut dessiner le résultat !

Première partie : grilles

Exercice 1

Le programme `CréerGrille(M, N)` renvoie une liste de listes, représentant une grille à M lignes et N colonnes, dont toutes les cases sont initialisées à la valeur `None`.

Exercice 2

La grille étant représentée par la liste `G` de ses lignes, le nombre de lignes est M `= len(G)`. Le nombre de colonnes est égal au nombre de cases de la première ligne (par exemple) c'est-à-dire N `= len(G[0])`. On en déduit le programme suivant.

```
def Taille(G) :
    M = len(G) ; N = len(G[0])
    return (M, N)
```

Exercice 3

Rappelons d'abord pourquoi `CG = G` ne marche pas pour créer une copie `CG` d'une liste (ou d'une liste de listes) `G`.

```
>>> G = CréerGrille(3, 5)
>>> CG = G
>>> G[0][0] = "Tût␣!"
>>> G
[['Tût␣!', None, None, None, None],
 [None, None, None, None, None],
 [None, None, None, None, None]]
>>> CG
[['Tût␣!', None, None, None, None],
 [None, None, None, None, None],
 [None, None, None, None, None]]
```

L'affectation `=` ne réalise pas une copie pour les « gros » objets comme les listes : on recopie seulement dans la variable `CG` l'adresse, en mémoire, de la liste `G`. Quand on modifie l'une des deux, on modifie la liste qui se situe à cette adresse, donc `G` et `CG` sont interchangeables.

On ne peut pas non plus utiliser la fonction de copie `CG = list(G)`, qui marcherait si `G` était une liste *sans* imbrication. Ici, c'est une liste de listes, donc il faut procéder différemment. Voici la solution la plus simple : on reprend le code de `CréerGrille`, mais on initialise les cases de la nouvelle grille avec la valeur `G[`i`][`j`]` plutôt que `None`.

```
def Copie(G) :
    (M, N) = Taille(G)
    return [[G[i][j] for j in range(N)] for i in range(M)]
```

Deuxième partie : cases aléatoires

Exercice 4

Pas de difficulté ici : on parcourt toute la grille, et on range dans une liste, quand on les rencontre, les cases vides.

```
def CasesVides(G) :
    (M, N) = Taille(G) ; V = []
    for i in range(M) :
        for j in range(N) :
            if G[i][j] == None :
                V.append( (i, j) )
    return V
```

Exercice 5

La fonction `EntAléa(a, b)` s'applique à deux entiers a et b et renvoie un entier, choisi aléatoirement selon la distribution uniforme, dans l'ensemble $\{a\,;\,a+1\,;\,\ldots\,;\,b-1\,;\,b\}$. Chaque valeur a donc la probabilité $1/(b-a+1)$ d'être choisie.

La fonction `Mélanger(L)` ne renvoie rien, mais modifie la liste `L` en permutant aléatoirement ses cases. Les $n!$ permutations (avec n la longueur de la liste) sont équiprobables.

```
>>> L = list(range(20)) ; L
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> Mélanger(L) ; L
[13, 4, 7, 0, 3, 15, 10, 9, 18, 19, 17, 14, 5, 16, 2, 8, 11, 6, 12, 1]
```

Exercice 6

1) Imaginons un tas de cartes, et qu'on doit en tirer aléatoirement n. Le plus simple est de mélanger le tas, puis de prendre les n premières (et pas, comme beaucoup l'ont imaginé, de remélanger le tas entre chaque pioche : ça ne sert à rien !). Appliqué à notre situation, on calcule la liste `V` des vases vides, on la mélange, et on renvoie les `nb` premières.

```
def ChoisirCasesVides(G, nb) :
    V = CasesVides(G)
    Mélanger(V)
    return V[ : nb]
```

2) Une fois ces cases « à murer » choisies, on y place des murs. Le programme ci-dessous ne renvoie rien : il modifie la grille `G`. Remarquons que celle-ci peut déjà contenir des murs ; dans ce cas le programme ci-dessous ajoute `nb` murs supplémentaires (donc distincts de ceux initialement présents).

```
def CréerMurs(G, nb) :
    Murs = ChoisirCasesVides(G, nb)
    for (i, j) in Murs :
        G[i][j] = "MUR"
```

Troisième partie : représentation graphique

Exercice 7

1) Si i désigne le numéro d'une ligne, il correspond géométriquement à une ordonnée (et pas une abscisse) ; tandis que si j désigne le numéro d'une colonne, il correspond géométriquement à une abscisse (et pas une ordonnée). Il est donc raisonnable de placer la case (i,j) aux coordonnées (j,i). Ceci place la ligne $i=0$ en bas ; si on préfère qu'elle se situe en haut, il suffit de prendre les coordonnées $(j, \mathrm{M}-1-i)$.

2) La fonction `fill` prend en argument deux listes : les abscisses de chaque point, et les ordonnées de chaque point. Et son rôle est de peindre l'intérieur du polygone ayant ces points pour sommets.

```
def DessinerCase(i, j, couleur) :
    fill([j - 0.5, j + 0.5, j + 0.5, j - 0.5, j - 0.5],
         [i - 0.5, i - 0.5, i + 0.5, i + 0.5, i - 0.5],
         color = couleur)
```

Exercice 8

D'abord quelques couleurs. On va faire le tour de la grille en noir, et les traits à l'intérieur en gris.

```
NOIR = (0.0, 0.0, 0.0)
GRIS = (0.7, 0.7, 0.7)
```

Pour dessiner une grille, on procède en trois étapes : d'abord on fait les $M-1$ lignes horizontales intérieures, puis les $N-1$ colonnes horizontales intérieures, puis enfin un rectangle (plus épais) pour le tour.

```
def DessinerGrille(M, N) :
    for i in range(0, M - 1) :
        plot([-0.5, N - 0.5],
             [i + 0.5, i + 0.5],
             linewidth = 1, color = GRIS)
    for j in range(0, N - 1) :
        plot([j + 0.5, j + 0.5],
             [-0.5, M - 0.5],
             linewidth = 1, color = GRIS)
    plot([-0.5, N - 0.5, N - 0.5, -0.5, -0.5],
         [-0.5, -0.5, M - 0.5, M - 0.5, -0.5],
         linewidth = 2, color = NOIR)
```

Exercice 9

Là il n'y a plus qu'à parcourir la grille à la recherche des murs, et utiliser la fonction `DessinerCase` pour les peindre en noir.

```
def DessinerMurs(G) :
    (M, N) = Taille(G)
    for i in range(M) :
        for j in range(N) :
            if G[i][j] == "MUR" :
                DessinerCase(i, j, NOIR)
```

Exercice 10

Un chemin est représenté par une liste de couples (i, j) désignant les cases que traverse ce chemin. Il faut en faire une liste d'abscisses (les j) et une liste d'ordonnées (les i), pour pouvoir utiliser la fonction `plot` qui tracera le trait matérialisant ce chemin.

```
def DessinerChemin(ch, couleur) :
    absc = [ch[k][1] for k in range(len(ch))]
    ords = [ch[k][0] for k in range(len(ch))]
    plot(absc, ords, linewidth = 3, color = couleur)
```

Quatrième partie : chemins

Exercice 11

Voilà une question de cours !

```
def VoisinesVides(G, i, j) :
    (M, N) = Taille(G) ; V = []
    for (vi, vj) in [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)] :
        if 0 <= vi < M and 0 <= vj < N and G[vi][vj] == None :
            V.append( (vi, vj) )
    return V
```

Exercice 12

Et ça c'est aussi, d'une certaine manière, une question de cours. En tout cas à connaître par cœur : pour choisir un élément au hasard d'une liste, on choisit un indice au hasard et on renvoie la valeur de la case correspondante.

```
def Choisir(L) :
    k = EntAléa(0, len(L) - 1)
    return L[k]
```

Exercice 13

Le programme ci-dessous s'arrête presque sûrement (une marche aléatoire, sur un ensemble fini, passe une infinité de fois sur toutes les cases accessibles), dès lors que les murs n'empêchent pas d'aller de la case de départ à la case d'arrivée.

```
def MarcheAléatoire(G, i0, j0, iF, jF) :
    ch = [(i0, j0)] ; i = i0 ; j = j0
    while (i, j) != (iF, jF) :
        (i, j) = Choisir(VoisinesVides(G, i, j))
        ch.append((i, j))
    return ch
```

Exercice 14

1) On cherche une boucle aussi grande que possible (tant qu'à faire !) du chemin : donc deux indices i (à partir de 0) et j (à partir de `len(L)` -1, c'est-à-dire la fin) tels que `L[i] == L[j]`.

```
def ChercherRépétition(L) :
    for i in range(len(L) - 1) :
        for j in range(len(L) - 1, i, -1) :
            if L[i] == L[j] :
                return (i, j)
    return None
```

2) Bon, on cherche une répétition avec le programme précédent, et on renvoie le chemin privé du tronçon correspondant.

```
def Simplifier(ch) :
    s = ChercherRépétition(ch)
    if s == None :
        return ch
    else :
        (k1, k2) = s
        return ch[ : k1] + ch[k2 : ]
```

3) Tant que le programme précédent ne renvoie pas le chemin de départ, on continue à chercher une répétition. Plutôt que de tester l'égalité des chemins (ce qui est long) on peut comparer leur longueur, c'est instané et c'est suffisant pour savoir si une simplification a été faite ou non.

```
def SimplifierAuMaximum(ch) :
    nouveau = Simplifier(ch)
    while len(nouveau) < len(ch) :
        ch = nouveau
        nouveau = Simplifier(ch)
    return nouveau
```

Exercice 15

1) Ici, pas grand chose à faire à part mettre bout à bout les programmes précédents.

```
def TrouverChemin(G, i0, j0, iF, jF) :
    ch = MarcheAléatoire(G, i0, j0, iF, jF)
    return SimplifierAuMaximum(ch)
```

2) Commençons par définir quelques couleurs : pour avoir les résultats comme dans l'énoncé, le point de départ sera magenta, le point d'arrivé sera vert, et le chemin sera jaune.

```
MAGENTA = (1.0, 0.0, 1.0)
VERT    = (0.0, 0.5, 0.0)
JAUNE   = (0.9, 0.7, 0.0)
```

Puis, dans cet ordre : on crée une grille, on y place des murs, on choisit deux cases vides (donc pas un des murs), la première des deux sera la case de départ, la seconde sera la case d'arrivée ; on calcule un chemin (simplifié au maximum) qui les relie, et on dessine.

```
>>> G = CréerGrille(50, 75)
>>> CréerMurs(G, 900)
>>> C = ChoisirCasesVides(G, 2)
>>> départ = (*C[0], MAGENTA)
>>> arrivée = (*C[1], VERT)
>>> ch = TrouverChemin(G, *C[0], *C[1])
>>> DessinerTout(G, [départ, arrivée], [(ch, JAUNE)])
>>>
```