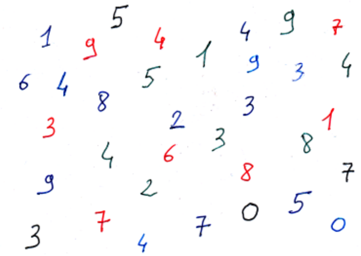
TP : Révisions de première année – Images et Graphes

Mise en situation



On se propose dans ce TP de manipuler une image couleur au format « png » sur laquelle sont dessinés des chiffres de 0 à 9. (voir image ci-contre). le but étant d'isoler chaque chiffre

Dans la 1ière partie nous allons modifier l'image pour se remémorer l'utilisation les fonctions numpy sur les tableaux pour les images.

Dans une 2nd partie nous allons construire un algorithme permettant d'isoler chacun des chiffres de manière à les afficher un à un.

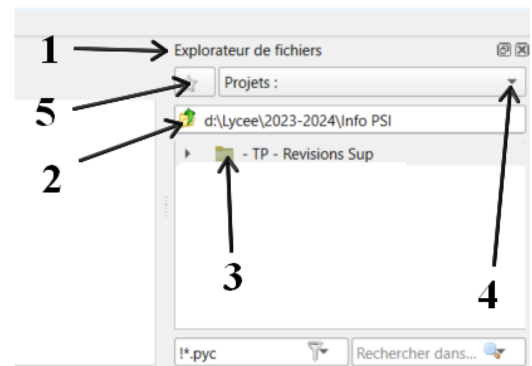
1ière Partie

Récupérations des données

Copier et enregistrer dans votre dossier personnel le dossier « TP Revisions Sup » qui contient les deux fichiers : « TP – Revisions.py » et « Chiffres_Couleur.png ». Puis ouvrir avec Pyzo le fichier « TP – Revisions.py ».

Le code fourni permet d'ouvrir l'image « Chiffres_Couleur.png » et de l'afficher avec la fonction « affiche(img) ». Cependant cela ne peut fonctionner que si le répertoire courant du Shell de Pyzo est le répertoire dans lequel se trouve le fichier image « Chiffres_Couleur.png ».

Procédure pour rendre courant le répertoire contenant le fichier « .png » :



- ☞ Visualiser la fenêtre de « l'explorateur de fichier » ou « File browser » dans la version anglaise. (Repère 1)
- ☞ Retrouver le dossier que vous avez copié dans vos propres données. Dossier où se situe le fichier « .png ». Pour cela naviguer dans les dossiers en utilisant la flèche verte pour revenir vers la racine du disque (Repère 2)
- ☞ Faire un clique droit sur votre dossier copié (Repère 3) et sélectionner « Ajouter ce répertoire à la liste des projets » ou « Star this directory » dans la version anglaise.
- ☞ Sélectionner votre dossier dans la liste déroulante des projets (Repère 4)
- ☞ Puis cliquer sur l'étoile (Repère 5) pour sélectionner « Aller dans ce dossier (Shell courant) » ou « Go to this directory in the curent Shell » dans la version anglaise.

Cette procédure doit lancer dans le Shell une commande permettant de modifier le répertoire du Shell courant. Ensuite l'exécution du code python du fichier « TP – Revisions.py » ne doit pas renvoyer d'erreur.

Question 1

A- Afficher l'image en lançant dans le Shell la commande : `>>> affiche(image_c)`

B- Lancer la commande affichant le pixel de la ligne 173 et de la colonne 119 de l'image :

```
>>> image_c[173,119]
```

C- Comment est codé un pixel de l'image ?

D- Quelle est le résultat obtenu par l'exécution des lignes 15 à 23 du code Python ?

E- Afficher l'image en niveau de gris (image_g) et afficher le pixel [173,119] de l'image en niveau de gris. Comment est codé un pixel de cette image en niveau de gris ?

Question 2

Ecrire une fonction `dim(img:np.ndarray) -> tuple` , qui prend en argument une image (tableau numpy à deux dimensions) et qui renvoie un couple d'entiers correspondant aux nombres de lignes et de colonnes de l'image.

Tester votre fonction avec l'image par la commande dans le Shell : `>>> dim(image_g)`

Question 3

Ecrire une fonction `img_blanche(nl:int,nc:int) -> np.ndarray`, qui prend en argument deux entiers nl et nc et qui renvoie une image (tableau numpy à deux dimensions) de nl lignes et nc colonnes. Attention : Les pixels de cette image doivent être des flottants et pas des entiers.

Tester votre fonction pour une image de 2 lignes et trois colonnes par la commande dans le Shell :

`>>> img_blanche(2,3)` Cela doit renvoyer un tableau 2×3 = 6 flottants de valeur : « 1. ».

Question 4

Ecrire une fonction `convertir_N_B(img:np.ndarray,seuil) -> np.array`, qui prend en argument une image en niveau de gris et un flottant compris entre 0.0 et 1.0 et qui renvoie un tableau numpy d'une image en noir ou blanc. C'est-à-dire une image ne contenant que des pixels noirs (Code 0.0) ou blancs (Code 1.0). Les pixels de l'image en niveau de gris dont le code est inférieur à la valeur du seuil seront convertis en pixels noirs, et les autres en pixels blancs.

Tester votre fonction avec l'image par la commande dans le Shell :

```
>>> image = convertir_N_B(image_g,0.75)
```

Vérifier votre fonction en affichant cette image par la commande dans le Shell :

```
>>> affiche(image)
```

Tester votre fonction avec deux autres valeurs de seuil : 0,3 et 0,9 par la commande dans le Shell :

`>>> affiche(convertir_N_B(image_g,0.3))` Conclure sur la valeur du seuil.

Question 5 (A ne faire que sur la première scéance)

Ecrire une fonction `sym_horiz(img:np.ndarray) -> np.array`, qui prend en argument une image en noir ou blanc et qui renvoie un tableau numpy d'une image en noir ou blanc.

L'image renvoyée sera une image symétrique à l'image prise en argument par rapport à un axe horizontal passant au milieu de l'image.

Tester votre fonction avec l'image par la commande dans le Shell :

```
>>> image_bis = sym_horiz(image_g)
```

Vérifier votre fonction en affichant cette image par la commande dans le Shell :

```
>>> affiche(image_bis)
```

Question 6 (A ne faire que sur la première scéance)

Ecrire une fonction `sym_verti(img:np.ndarray) -> np.array`, qui prend en argument une image en noir ou blanc et qui renvoie un tableau numpy d'une image en noir ou blanc.

L'image renvoyée sera une image symétrique à l'image prise en argument par rapport à un axe vertical passant au milieu de l'image.

Tester votre fonction avec l'image par la commande dans le Shell :

```
>>> image_bis = sym_verti(image)
```

Vérifier votre fonction en affichant cette image par la commande dans le Shell :

```
>>> affiche(image_bis)
```

Question 7 (A ne faire que sur la première scéance)

Ecrire une fonction **`rot_gauche(img:np.ndarray) -> np.array`**, qui prend en argument une image en noir ou blanc et qui renvoie un tableau numpy d'une image en noir ou blanc.

L'image renvoyée sera la rotation de 90° dans le sens trigonométrique de l'image prise en argument. Tester votre fonction avec l'image par la commande dans le Shell :

```
>>> image_bis = rot_gauche(image)
```

Vérifier votre fonction en affichant cette image par la commande dans le Shell :

```
>>> affiche(image_bis)
```

Question 8 (A ne faire que sur la première scéance)

Ecrire une fonction **`rot_180(img:np.ndarray) -> np.array`**, qui prend en argument une image en noir ou blanc et qui renvoie un tableau numpy d'une image en noir ou blanc.

L'image renvoyée sera la rotation de 180° de l'image prise en argument. Votre fonction devra utiliser une ou plusieurs des fonctions définies aux questions 5, 6 et 7.

Tester votre fonction avec l'image par la commande dans le Shell :

```
>>> image_bis = rot_180(image)
```

Vérifier votre fonction en affichant cette image par la commande dans le Shell :

```
>>> affiche(image_bis)
```

Question 9 (A ne faire que sur la première scéance)

Ecrire une fonction **`rot_droite(img:np.ndarray) -> np.array`**, qui prend en argument une image en noir ou blanc et qui renvoie un tableau numpy d'une image en noir ou blanc.

L'image renvoyée sera la rotation de –90° de l'image prise en argument. Votre fonction devra utiliser une ou plusieurs des fonctions définies aux questions 5, 6, 7 et 8.

Tester votre fonction avec l'image par la commande dans le Shell :

```
>>> image_bis = rot_droite(image)
```

Vérifier votre fonction en affichant cette image par la commande dans le Shell :

```
>>> affiche(image_bis)
```

2ième Partie

Récupérations des données

Copier et enregistrer dans votre dossier personnel le dossier « TP Revisions Sup 2 » qui contient les trois fichiers : « TP – Revisions 2.py », « Chiffres_Couleur.png » et « Exemple.png ».

Puis ouvrir avec Pyzo le fichier « TP – Revisions 2.py » qui contient le code de toutes les questions de la première partie. Soit le corrigé de cette première partie.

Les lignes de code 178, 179 et 180 de ce code permettent de définir trois images de 4×4 = 16 pixels.

- ☞ L'image en couleur « exemple_c » n'a que des pixels blanc, noir, vert ou rouge.
- ☞ L'image en niveau de gris « exemple_g » n'a que des pixels blanc, noir, gris clair ou gris foncé.
- ☞ L'image noir ou blanc « exemple » n'a que des pixels noirs ou blancs.

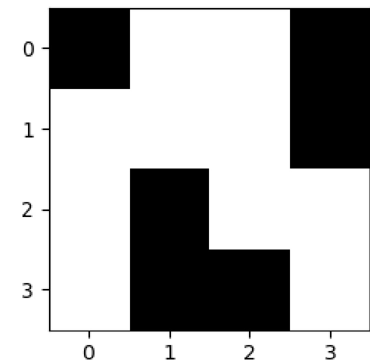
Chacune des ces trois images peut être visualisée par la commande Shell :

```
>>> affiche(exemple_c)     >>> affiche(exemple_g)     >>> affiche(exemple)
```

Tester cela après avoir défini comme répertoire du Shell courant le dossier que vous avez copié. Voir pour cela la procédure décrite en page 1 de cet énoncé.

Pour la suite nous commençons avec l'image « exemple » qui est en noir ou blanc.

Principe de l'algorithme



On a donc l'image ci-contre (4×4 = 16 pixels) qui est en noir ou blanc. Le but est de créer un graphe dont les sommets sont les pixels noirs et les arrêtes les relations de voisinage entre ces pixels noirs.

Deux pixels sont voisins s'ils partagent une arête ou un sommet du carré qui les définit. Donc un pixel noir ne peut avoir au maximum que 8 voisins voir 5 ou 3 si il est sur le bord ou dans le coin de l'image.

Nous voyons ici que le graphe correspondant à cette image a trois composantes connexes. Le pixel noir en haut à gauche, les deux pixels noirs en haut à droite, et les trois pixels noirs sur les deux lignes du bas.

Le graphe sera décrit par un tableau numpy dont les dimensions sont celles de l'image. Les valeurs de ce tableau numpy est l'ensemble vide (**None** en langage Python) si le pixel correspondant est blanc. Ou alors, si le pixel est noir, la valeur est un couple constitué d'un indice et de la liste des voisins noirs de ce pixel. L'indice, un entier, sera le même pour chaque pixel d'une même composante connexe.

Ainsi le graphe correspondant à cette image sera le tableau numpy ci-dessous.

```
array([[(0, []), None, None, (1, [(1, 3)])],
       [None, None, None, (1, [(0, 3)])],
       [None, (2, [(3, 1), (3, 2)]), None, None],
       [None, (2, [(2, 1), (3, 2)]), (2, [(2, 1), (3, 1)]), None]],    dtype=object)
```

La composante connexe en haut à gauche a l'indice 0 celle en haut à droite l'indice 1 et celle en bas l'indice 2. Les questions 10, 11, 12, 13 et 14 vont nous permettre de construire ce graph.

Les questions 15, 16 et 17 permettront d'afficher successivement trois images ne comportant chacune que les pixels correspondant à une seule composante connexe de ce graphe.

Cette procédure appliquée à l'image des chiffres (voir page 1 de l'énoncé) permettra d'isoler chaque chiffre de l'image pour les afficher successivement de haut en bas de l'image.

Question 10

Ecrire une fonction **graphe_vide(nl:int,nc:int)->np.ndarray**, qui prend en argument deux entiers et qui renvoie un graphe vide correspondant à une image ayant nl lignes et nc colonnes et qui correspondrait à une image n'ayant que des pixels blancs.

Tester votre fonction avec l'image par la commande dans le Shell : **>>> graphe_vide(4,4)**

Celle-ci doit retourner le tableau numpy ci-dessous :

```
array([[None, None, None, None],
       [None, None, None, None],
       [None, None, None, None],
       [None, None, None, None]], dtype=object)
```

Question 11

La fonction **liste_voisins(imgnb:np.ndarray,l:int,c :int)->list**, prend en argument une image en noir ou blanc (tableau numpy), et deux entiers **l** et **c** qui sont les indices de ligne et de colonne d'un pixel. Cette fonction retourne la liste des couples indices des pixels noirs voisins du pixel **(l,c)**. Justifier l'utilisation des fonctions max et min dans les lignes 195 et 196.

Compléter le code de cette fonction **liste_voisins(imgnb:np.ndarray,l:int,c :int)**. Puis tester votre fonction part les commandes dans le shell :

```
>>> liste_voisins(exemple,0,0)     >>>      liste_voisins(exemple,1,3)
                >>> liste_voisins(exemple,3,2)
```

Ces commandes doivent renvoyer respectivement :

```
[]                    [(0,3)]                    [(2,1),(3,1)]
```

Question 12

Compléter la fonction **`creer_graph(img:np.ndarray,seuil:float)->np.ndarray`**, qui prend en argument une image en niveau de gris (tableau numpy) un seuil de niveau de gris (un flottant compris entre 0.0 et 1.0) et qui renvoie un graphe (tableau numpy) correspondant à l'image passée en noir ou blanc avec des indices qui restent cependant non définie (**`None`**). Utiliser dans cette fonction, la fonction **`convertir_N_B(img:np.ndarray,seuil:float) -> np.ndarray`** de la première partie, ainsi que la fonction **`liste_voisins(imgnb:np.ndarray,l:int,c :int)->list`** de la question précédente

Tester la fonction par la commande dans le Shell : **`>>> creer_graph(exemple_g,0.7)`**

Celle-ci doit retourner le tableau numpy ci-dessous :

```
array([[(None, []), None, None, (None, [(1, 3)])],
       [None, None, None, (None, [(0, 3)])],
       [None, (None, [(3, 1), (3, 2)]), None, None],
       [None, (None, [(2, 1), (3, 2)]), (None, [(2, 1), (3, 1)]), None]],  dtype=object)
```

Question 13

Compléter la fonction **`indicage_pixels(graph:np.ndarray,i:int,j:int,ind:int)->None`**, qui prend en argument un graphe (tableau numpy), les coordonnées **`l`** et **`c`** d'un pixel (2 entiers) et un indice (un entier) et qui (si le pixel **`(l,c)`** n'est pas indicé) indice (avec l'entier **`ind`**) le pixel **`(l,c)`** ainsi que tous les pixels de la composante connexe du graph à laquelle le pixel appartient.

L'indiçage de tous les autres pixels de la composante connexe sera effectué par un parcours en profondeur de la composante connexe du graphe. Ce parcours en profondeur sera réalisé par un appel récursif de cette fonction **`indicage_pixels`**, en indiçant les pixels voisins.

Tester avec l'image par les commandes : **`>>> graph=creer_graph(exemple_g,0.7)`**

puis : **`>>> indicage_pixels (graph,2,1,2)`** puis : **`>>> graph`**

La troisième commande doit retourner le tableau numpy ci-dessous :

```
array([[(None, []), None, None, (None, [(1, 3)])],
       [None, None, None, (None, [(0, 3)])],
       [None, (2, [(3, 1), (3, 2)]), None, None],
       [None, (2, [(2, 1), (3, 2)]), (2, [(2, 1), (3, 1)]), None]],  dtype=object)
```

Ainsi la composante connexe du bas de l'image est indicée avec l'indice 2.

Question 14

Pour indicer entièrement le graphe d'une image de grande taille avec beaucoup de composantes connexes, indicer une à une et manuellement ses composantes connexes n'est pas envisageable. Il faut donc pouvoir indicer automatiquement toutes les composantes connexes d'un graph non indicé avec un indice différent pour chaque composante connexe. C'est l'objectif de la fonction suivante.

Compléter la fonction **`indicage_graph(graph:np.ndarray)->int`**, qui prend en argument un graphe (non indicé) et qui indice tous les pixels noirs du graphe avec un indice différent pour chaque composante connexe de ce graphe. Fonction qui retournera le nombre d'indices différents du graphe.

Pour cela il faut parcourir le graphe (l'image) de haut en bas et de gauche à droite pour dénicher les pixels noirs non coloriés et attribuer à ce pixel et tous ceux de la composante connexe un indice propre à la composante connexe. Pour cela on utilise la fonction **`indicage_pixels`** de la question précédente.

Tester avec l'image par les commandes : **`>>> graph=creer_graph(exemple_g,0.7)`**

puis : **`>>> indicage_graph(graph)`** puis : **`>>> graph`**

La seconde commande doit retourner l'entier **`3`** et la troisième le tableau numpy ci-dessous :

```
array([[(0, []), None, None, (1, [(1, 3)])],
       [None, None, None, (1, [(0, 3)])],
       [None, (2, [(3, 1), (3, 2)]), None, None],
       [None, (2, [(2, 1), (3, 2)]), (2, [(2, 1), (3, 1)]), None]],  dtype=object)
```

Question 15

Il est possible de tester la fonction précédente avec l'image en niveau de gris des chiffres : Taper les commandes **`>>> graph=creer_graph(image_g,0.7)`** puis : **`>>> indicage_graph(graph)`** puis : **`>>> graph`** Cependant cette fois, comme le graphe est de grande taille la troisième commande n'affiche dans le Shell qu'une petite partie du graphe (les pixels situés dans les coins). Donc on ne sait pas si la commande a bien fonctionné.

Pour le vérifier il faut taper la commande : **`>>> graph[115,260]`**

Cela renvoi : **`(15, [(114, 261), (115, 259), (115, 261), (116, 259), (116, 260), (116, 261)])`** Ce qui tend à prouver que la fonction a bien fonctionné car le pixel (115,260) appartient au 16ième chiffre de l'image en partant du haut, (indice15) qui est un chiffre « 3 ». Ensuite la liste des pixels de la liste sont bien des voisins du pixel (115,260).

Nous allons donc, à partir du graphe colorié précédemment, créer une image (aux dimensions de l'image en niveau de gris) affichant ce chiffre « 3 ». C'est l'objectif de la fonction suivante.

Compléter la fonction **`trace_chiffre(graph:np.ndarray,ind:int)->np.ndarray`**, qui prend en argument un graphe indicé (tableau numpy) et l'indice d'un chiffre (entier) et qui renvoi une image aux dimensions du graphe (tableau numpy). Cette image sera une image, à la dimension du graphe, dont le fond est blanc et sur lequel est tracé en noir le chiffre correspondant à l'indice.

Tester avec l'image des chiffres par : **`>>> graph=creer_graph(image_g,0.7)`**

puis : **`>>> indicage_graph(graph)`**

puis : **`>>> affiche(trace_chiffre(graph,15))`**

Cette dernière commande doit afficher le chiffre « 3 » isolé (16ième chiffre dont l'indice est 15).

Question 16

A cette question on ne demande aucun codage python. Il s'agit juste de vérifier que l'ensemble des fonctions fonctionne correctement.

Tester l'ensemble des fonctions avec l'image des chiffres par la commande :

```
>>> animation_chiffres(image_g,0.7)
```

Tester avec l'image des chiffres et deux autres valeurs de seuil par les commandes :

```
>>> animation_chiffres(image_g,0.3)
>>> animation_chiffres(image_g,0.9)
```

Conclure

Question 17 (Pour les plus rapides)

On envisage d'utiliser ce code pour faire de la reconnaissance de caractères. Pour cela il est préférable d'utiliser des images dont les dimensions sont celles du chiffre isolé sans tout l'espace blanc autour.

Compléter la fonction **`animation_zoom(img:np.ndarray,seuil :float)->None`**, qui prend en argument une image en niveau de gris et affiche successivement les composantes connexes (chiffres) dans une image ayant les dimensions de cette composante connexe (dimension du chiffre).

Vous remarquerez que le code donné est identique à celui de la fonction **`animation_chiffres`** sauf la ligne 306 où on a la commande **`affiche(diapo_zoom)`** en lieu et place de **`affiche(diapo)`**.

Donc le code à compléter entre les commandes **`diapo=trace_chiffre(graph,i)`** et **`affiche(diapo_zoom)`** permet juste de créer, à partir de l'image du chiffre isolé (diapo) dans une grande image blanche, une autre image (diapo_zoom) plus petite (la plus petite possible) qui peut contenir le chiffre isolé.

Le reste du code comme dans la fonction permet d'animer un affichage successif de tous les chiffres de l'image originelle.