Université de Strasbourg
Première année de licence
Semestre 1

Algorithmique et Programmation 1

TP2 : Les entrées/sorties et les boucles

À partir de ce TP, nous vous demandons d'utiliser exclusivement le mode script. Si vous n'avez pas eu le temps de configurer *Geany* (ou un autre IDE), veuillez revoir la section 4 du TP précédent.

Par ailleurs, pour chacune des fonctions ci-dessous, prévoir leur comportement pour différentes valeurs des arguments : une fois que la fonction est écrite, tester chaque fonction pour ces valeurs d'arguments et vérifier si ces fonctions se comportent bien de la façon prévue.

1 Dessiner un sapin

Nous cherchons à dessiner un sapin comme sur la figure 1. Malgré les apparences, ce n'est pas une tâche triviale. Une des compétences importantes de l'informaticien est celle de pouvoir décomposer une tâche complexe en plusieurs tâches simples et de les résoudre séparément quand c'est possible. Pour l'heure nous vous proposons une décomposition possible.

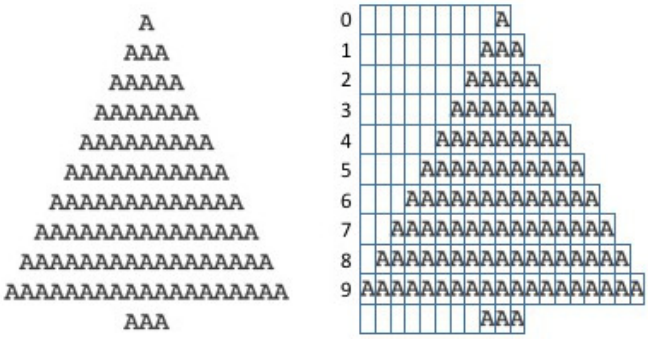

FIGURE 1 – On cherche à représenter un sapin (figure de gauche). Pour cela, on crée des chaînes de caractères composées d'espaces et de `'A'` (schématisé sur la figure de droite).

Lancer votre IDE *(Geany)*. Créer un nouveau fichier et l'enregistrer sous un nom se terminant par `.py` (par exemple `sapin.py`). Écrire et tester les fonctions suivantes dans ce fichier.

1. Écrire une fonction `a_la_chaine` qui admet un argument entier `n`. La valeur de retour de cette fonction est une chaîne de caractères composée d'une juxtaposition de `n` caractères `A`. Par exemple `a_la_chaine(5)` devra être la chaîne `"AAAAA"`.
2. Écrire une fonction `a_la_chaine2` qui admet deux arguments entiers `esp` et `n`. Cette fonction devra retourner une chaîne de caractères composée de `esp` espaces suivis de `n` caractères `A`. Par exemple la valeur de `a_la_chaine2(2,7)` devra être la chaîne `"  AAAAAAA"`.

3. Écrire une fonction `colonne` qui admet un argument entier `n`. Cette fonction ne retourne rien (retourne `None`). Mais elle devra afficher 10 fois une chaîne de caractères composée de `n` fois `'A'`. Par exemple l'appel `colonne(5)` devra afficher ce qui est représenté à gauche de la figure 2. Pour écrire cette fonction, vous pouvez avoir recours aux fonctions que vous avez déjà écrites ci-dessus.
4. Écrire une fonction `diagonale1` qui admet un argument entier `n`. La fonction devra afficher `n` lignes. La première ligne devra contenir 1 caractère `A`. La deuxième ligne devra contenir 2 caractères `A` et ainsi de suite jusqu'à ce que les `n` lignes aient été écrites. L'appel `diagonale1(10)` devrait produire la sortie représentée sur la figure 2 au milieu.

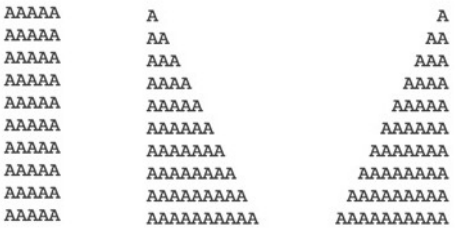

FIGURE 2 – Exercices préparatoires au dessin du sapin

5. Écrire une fonction `diagonale2` qui admet un argument entier `n`. La fonction devra afficher `n` lignes. Toutes les lignes contiennent `n` caractères. Sur la première ligne, il doit y avoir des espaces sauf un caractère `A` à la dernière position. Sur la deuxième ligne, il doit y avoir deux caractères `A` à la fin et ainsi de suite jusqu'à ce que les `n` lignes aient été écrites. L'appel `diagona2(10)` devrait produire la sortie représentée sur la figure 2 à droite. La principale difficulté est de déterminer le nombre d'espaces et de caractères `A` à écrire sur chaque ligne.
6. Écrire une fonction `sapin` qui prend en argument un nombre de lignes `n`. Cette fonction devra afficher `n` lignes. La première devra contenir un caractère `A`. La deuxième devra en contenir 3 et ainsi de suite jusqu'à ce que les `n` lignes aient été écrites. Pour finir le programme devra dessiner une ligne contenant trois caractères `A` et représentant le tronc du sapin. L'appel `sapin(10)` devra produire la sortie représentée sur la figure 1. Tester cette fonction pour plusieurs valeurs de `n` et vérifier que les résultats sont conformes à vos attentes.

2 Projet labyrinthe : afficher des images

Pour afficher des images dans une fenêtre, nous allons utiliser le module `tkdraw.basic` qui à son tour utilise la bibliothèque python `tkinter`.

2.1 Le module `tkdraw.basic`

Pour commencer, installez le package `tkdraw` sur votre machine. Sur les machines de l'université, il suffit de taper les commandes suivantes dans un terminal (à effectuer une seule fois sur votre compte) :

```
python3 -m pip install --upgrade pip
python3 -m pip install tkdraw
```

Note. Pour installer ce package sur votre machine personnelle, il faut vous assurer d'avoir installé pip et tkinter auparavant. Ce deux commandes font cela sur une distribution Linux standard (Debian ou Ubuntu) :

```
sudo apt-get update
sudo apt-get install python3-pip python3-tk
```

Une aide pour le module tkdraw.basic est disponible en ligne sous python, en tapant les commandes suivantes dans un terminal :

```
$ python3
>>> import tkdraw.basic as graph
>>> help(graph)
```

La première image : une fenêtre blanche

Créer, avec *Geany*, un fichier `TP2_img.py` et y écrire le programme suivant :

```
import tkdraw.basic as graph

graph.open_win(120, 160)
graph.wait()
```

1. La première ligne (`import tkdraw.basic as graph`) charge le module ;
2. La deuxième ligne (`graph.open_win(120, 160)`) ouvre une fenêtre et crée une image de 120 pixels de haut et 160 pixels de large. Les pixels de cette image sont initialement blancs (figure 3.a) ;
3. La troisième ligne (`graph.wait()`) attend que l'utilisateur ferme la fenêtre, puis le programme se termine. Notez que si vous n'appelez pas la fonction `wait` le programme se termine immédiatement et la fenêtre se ferme à ce moment, juste après avoir été ouverte.

Deuxième étape : dessiner un point

```
import tkdraw.basic as graph

graph.open_win(120, 160)
graph.plot(20, 30)
graph.wait()
```

Nous avons ajouté une seule instruction à la troisième ligne : `graph.plot(20, 30)`. Cette instruction a pour effet de colorier en noir le point d'ordonnée 20 et d'abscisse 30. Vous l'aurez noté : l'ordonnée 0 ne correspond au bord inférieur de l'image, mais au bord supérieur (figure 3.b). Assurez-vous que vous obtenez le même résultat.

Troisième étape : dessiner une ligne

```
import tkdraw.basic as graph

graph.open_win(120, 160)

for x in range(160):
   graph.plot(40, x)

graph.wait()
```

À présent, l'instruction `graph.plot(40, x)` est dans une boucle `for` qui fait varier la valeur de `x` de 0 jusqu'à 159 (toute la largeur de l'image). Ainsi, on obtient une ligne noire parcourant toute la largeur de l'image et située à 40 pixels du bord supérieur (figure 3.c). Assurez-vous que vous obtenez le même résultat.

Quatrième étape : en faire des fonctions

Voici le même programme mais avec une fonction `ligne_horiz` qui prend en argument l'ordonnée à laquelle on souhaite placer la ligne, et la largeur de l'image. On suppose que l'ordonnée est une valeur valide (`0 <= y < 120` dans notre cas).

```
import tkdraw.basic as graph

def ligne_horiz(y, larg):
   for x in range(larg):
      graph.plot(y, x)
```

```
##### programme principal #####
graph.open_win(120, 160)
ligne_horiz(40, 160)
graph.wait()
```

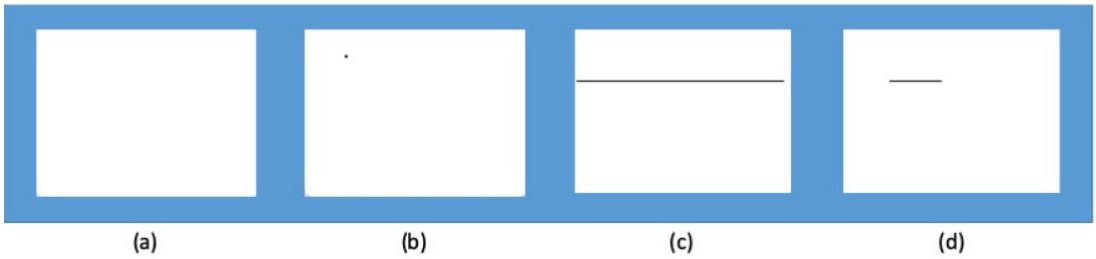

FIGURE 3 – Une fenêtre uniformément blanche (a) un point noir (b) une ligne (c)

2.2 À vous de jouer

Dans les exercices suivants, il s'agira d'écrire d'autres fonctions, sans supprimer les précédentes (il suffira de ne pas les appeler si vous n'en avez pas besoin). Nous vous demandons de ne pas utiliser les conditionnelles (`if`, `else`, `while`), mais de vous inspirer de ce que vous avez déjà fait pour le sapin.

1. Écrire une fonction `segment_horiz` qui est très semblable à la fonction `ligne_horiz`. Cette fonction admet trois entiers en arguments : `y`, `x1`, `x2`. Au lieu de dessiner une ligne horizontale sur toute la largeur de l'image, cette fonction devra se limiter à la partie de la ligne comprise entre les abscisses `x1` et `x2`. On suppose que `0 <= x1 <= x2 < 160`. Par exemple `segment_horiz(40,40,60)` devrait donner quelque chose de semblable à la figure 3.d.
2. Écrire une fonction `rectangle` qui admet comme arguments quatre entiers : `y1, y2, x1` et `x2`. Cette fonction devra colorier en noir les pixels dont l'abscisse est comprise entre `x1` et `x2` et dont l'ordonnée est comprise entre `y1` et `y2`. Là aussi, on suppose implicitement que `0 <= x1 < x2 < 160` et `0 <= y1 < y2 < 120`. Par exemple `rectangle(20,60,40,60)` devrait donner quelque chose de semblable à la figure 4.a. *Indice :* vous savez déjà dessiner des segments entre `x1` et `x2`. Il suffit d'en dessiner autant qu'il y a de lignes entre `y1` et `y2`.
3. Écrire une fonction `rayures_vertic` qui admet comme arguments trois entiers : `haut` la hauteur de l'image, `larg` la largeur de l'image, et `larg_bande`. On suppose que la largeur de l'image `largeur` est un multiple de `2*larg_bande`. Cette fonction produit une image composée de rayures verticales noires et blanches de largeur `larg_bande`. On demande à ce que la première rayure soit blanche. Par exemple `rayures_vertic(120,160,20)` devrait donner quelque chose de semblable à la figure 4.b. *Indice :* les rayures ne sont que des rectangles. Vous pouvez donc utiliser la fonction `rectangle` écrite ci-dessus.

2.3 Pour aller plus loin (difficile)

1. Écrire une fonction `damier` qui admet comme arguments trois entiers : `haut` la hauteur de l'image, `larg` la largeur de l'image, et `cote`. On suppose que `larg` et `haut` sont tous les deux des multiples de `2 * cote`. Cette fonction devra produire un damier dont la largeur des cases est de `cote`. Par exemple `damier(120,160,20)` devrait donner quelque chose de semblable à la figure 4.c. Là aussi, ne pas hésiter à faire appel à la fonction `rectangle`.

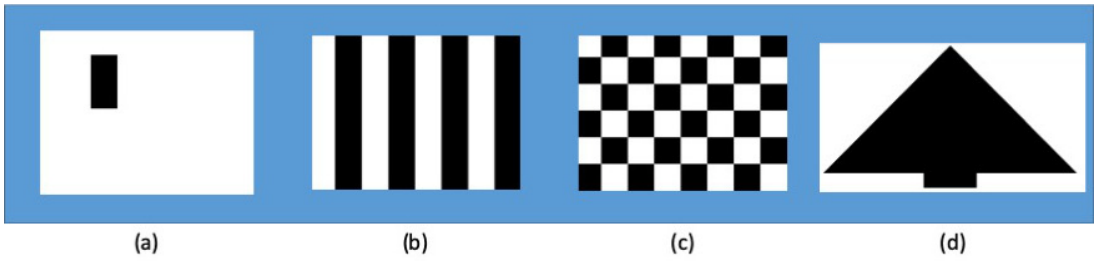

FIGURE 4

2. Enfin, écrire une fonction `sapin` qui admet deux arguments entiers : `haut` la hauteur de l'image, et `larg` la largeur de l'image. Sur la première ligne, il ne faudra colorier qu'un seul pixel au milieu du bord supérieur de l'image. Sur la deuxième ligne, il faudra colorier 3 pixels. Sur la troisième ligne 5 pixels, et ainsi de suite. On arrêtera cet algorithme une fois arrivé aux 9 dixièmes de la hauteur. Sur les lignes suivantes, on représente le tronc d'une largeur de 40 pixels. N'essayez pas de faire intervenir la fonction rectangle, sauf peut-être pour le tronc. La figure 4.d a été obtenue pour `haut=110` et `larg=200`.