université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

semaine du 2 octobre 2023

Introduction à l'informatique
L1

TP n° 4

Consignes les exercices ou questions marqués d'un ⋆ devront être rédigés sur papier (afin de se préparer aux épreuves écrites du partiel et de l'examen). En particulier, il est recommandé d'être dans les même conditions qu'en examen : pas de document ni de calculatrice. Les questions marquées d'un ⋄ sont des questions supplémentaires permettant d'aller plus loin (et ne seront pas forcément corrigées en TP). Tous les TPs se font sous Linux.

Exercices

Exercice 0

Ouvrir un terminal :
— créer un répertoire TP4 à l'intérieur du répertoire `IntroInfo`
— se placer à l'intérieur du répertoire `TP4`
On suppose pour les autres exercices que le répertoire `TP4` est le répertoire courant.

Exercice 1

⋆ On considère le programme suivant :

```
1 tab = [0] * 11
2 for i in range(len(tab)):
3    if i % 2 == 1:
4        tab[i] = tab[i-1]
5    else:
6        tab[i] = i
7 print(str(tab))
```

1. Décrire succinctement ce que fait chaque ligne.
2. Qu'affiche la dernière ligne ?

Réponse:

1. Pour chaque ligne :
   (1) définit une variable `tab` contenant un tableau de 11 cases valant toutes 0.
   (2) commence une boucle **`for`**, la variable `i` prend les valeurs 0 à 10 (`range` renvoie les valeurs entre 0 inclus à 11 exclus).
   (3) teste si la valeur courante de `i` est impaire.
   (4) si oui, copie la case précédente dans la case d'indice `i`
   (5) sinon
   (6) mets la valeur `i` dans la case `i`.
   (7) après la boucle, affiche le tableau `tab`.
2. Affiche `[0, 0, 2, 2, 4, 4, 6, 6, 8, 8, 10]`

Exercice 2

Écrire un programme Python ayant la structure suivante :

```
1  tab1 = [ 1, 2, 3, 4, 5, 6 ]
2  tab2 = [ 10, 20, 100, 1000 ]
3  tab3 = [ 5, 10, 7, 34, 35, 36 ]
4  tab4 = [ ]
5  tab5 = [ 10, 10, 100, 1000 ]
6  tab6 = [ 10, 10, 100, 1000, 1 ]
7
8  tab = tab1
9
10 #Completer ci-desssous.
11 #...
```

Le code définit 6 tableaux. La ligne 8 définit la variable `tab` comme référençant le tableau `1`. Compléter le code à l'endroit indiqué pour vérifier que le tableau `tab` est trié. On doit respecter les contraintes suivantes :

- si le tableau est trié, on veut afficher `Trié`
- si le tableau n'est pas trié, on veut afficher `Non trié`
- on souhaite s'arrêter au premier indice pour lequel le tableau n'est pas trié. Par exemple pour le tableau `tab3`, on souhaite ne parcourir le tableau que jusqu'au troisième élément

Tester ensuite votre programme en modifiant la ligne 8 en `tab = tab2`, `tab = tab3`, *etc.*

Réponse: On peut guider plus ou moins en fonction de l'avancement du groupe.

- D'abord demander à l'oral quels tableaux sont triés dans les exemples
- Réfélchir au type de boucle à utiliser
- Une fois que le code est écrit montrer comment le simplifier
- Rappel : `break` et `continue` n'ont pas encore été vus, mais le seront plus tard.

Exercice 3

La conjecture de Collatz peut s'énoncer de la façon suivante (informellement) :

- choisir un entier n strictement positif.
- si n est pair, le diviser par deux, sinon calculer $3 \times n + 1$

Répéter ces opérations sur le résultat obtenu. On arrive toujours à une suite se terminant par 4, 2 et 1. (Et une fois qu'on est à 1 on « bouclera » toujours sur $3 * 1 + 1 = 4 \to \frac{4}{2} = 2 \to 1 \to 4 \to 2 \to 1 \to \ldots$)

1. définir une variable `n` contenant un entier strictement positif.
2. afficher le nombre d'étapes que prend la suite d'opérations ci-dessus pour arriver à 1. **Attention** il faut bien utiliser l'opérateur de division entière `//` sans quoi, le programme sera fortement buggué.
3. modifier le programme pour afficher la plus grande valeur atteinte par la suite, pour l'entier `n` choisi au départ.
4. ◇ modifier le programme pour qu'il cherche, entre 1 et 100 la valeur de n ayant le plus grand nombre d'étapes.

Exercice 4

Le crible d'Ératosthène est une méthode pour calculer tous les nombres premiers inférieurs à une borne n donnée. On considère le code Python ci-dessous :

```
1  n = 1000 #On veut trouver tous les nombres premiers inferieurs a 1000
2
3  tab = n * [True]
4  tab[0] = False   #0 n'est pas premier
5  tab[1] = False   #1 n'est pas premier
6  #Completer ci-dessous.
```

Compléter le code à l'endroit indiquer pour effectuer le crible sur le tableau `tab`. L'algorithme à appliquer est le suivant :

- pour tout entier i compris entre 2 (inclus) et n (exclus)
- si `tab[`i`]` est vrai alors :
  - pour tous les j, multiples de i entre $2 \times i$ (inclus) et n (exclus) :
    - mettre `tab[`j`]` à faux

Après ce traitement, les indices de toutes les cases valant vrai sont des nombres premiers. Par exemple :

- On commence à parcourir avec i valant 2. `tab[2]` est vrai. On parcourt donc tout le tableau pour des valeurs de j valant 4, 6, 8, ...et on marque toutes ces cases à faux (ce sont des multiples de 2, donc pas des nombres premiers).
- On passe à i valant 3. `tab[3]` est vrai. On parcourt donc tout le tableau pour des valeurs de j valant 6, 9, 12, ...et on marque toutes ces cases à faux.
- On passe à i valant 4. `tab[3]` est faux (marqué lors du traitement des multiples de 2), on passe à la suite.
- On passe à i valant 5. `tab[5]` est vrai. On parcourt donc tout le tableau pour des valeurs de j valant 10, 15, 20, ...et on marque toutes ces cases à faux.
- ...

Réponse: Code basique dans `exo4.py`. C'est une traduction directe de l'algorithme. Dans la boucle interne, on utilise **`range`** avec trois paramètres pour avoir un « pas » différent de 1 (cf. cours). On évitera d'écrire **`if`** `tab[i] == True:` qui est redondant.

◇ Dans l'exemple ci-dessus, on remarque que certains nombres sont marqués plusieurs fois à faux (par exemple 6 et 12 sont marqués lors du parcours du 2 et du 3).

1. ◇ Pour i valant 2, 3, 5, 7 : quelle est la première valeur de i qui ne vaut pas déjà faux lorsqu'on parcourt les multiples de i.
2. ◇ En déduire une modification du parcours des multiples évitant de remarquer plusieurs fois une valeur à faux.
3. ◇ En vous servant de l'observation précédente, est-il nécessaire de parcourir tous les entiers entre 2 et n ? peut-on s'arrêter avant ?

Réponse: Pour les questions supplémentaires :

1. pour 2, la première valeur est 4, pour 3, la première valeur est 9 (6 est marqué par le parcours des multiples de 2). Pour 5, la première valeur est 25 (10 et 20 sont marqués par les multiples de 2, 15 par les multiples de 3). Pour 7, la première valeur est 49 (14, 28, 42 sont des multiples de 2, 21 est un multiple de 3 et 35 est un multiple de 5).
2. En généralisant, on remarque que tous les multiples de i inférieurs à i^2 sont déjà marqué. La boucle interne devient : **`for`** `j` **`in`** **`range`**`(i*i, n, i):`.
3. Puisqu'on parcourt, dans la boucle interne, à partir de i^2, on se rend compte que dès qu'on dépasse $i = \sqrt{n}$, alors i^2 dépasse n. On peut donc arrêter la boucle externe à $\sqrt{n}$. La fonction `sqrt` est accessible dans le module `math` :

```
1 from math import sqrt
2
3 #...
4
5 for i in range(2, int(sqrt(n))+1):
6 #...
```

Le code est dans le fichier `exo4_bis.py`

Exercice 5

On souhaite utiliser de nouveau la bibliothèque `turtle` (cf feuille 3 pour les commandes).

1. Écrire un programme Turtle demandant à l'utilisateur un entier `n` et traçant un escalier à `n` marches. On utilisera les instructions `forward`/`left` et `right` plutot que des coordonnées absolues. Une marche fait 30 pixels de largeur et 30 pixels de hauteur.

Réponse:

```
n = int(input("Nombre de marches : "))
down()
for _ in range(n):
    left(90)
    forward(10)
    right(90)
    forward(10)
up()
done()
```

2. Écrire un programme Python qui dessine 5 carrés noirs de 30 pixels de côté, alignés horizontalement et séparés chacun de 30 pixels.
3. ◇ S'inspirer du code précédent pour dessiner un damier de 5 cases de côté.

Réponse: Il y a plusieurs façons de faire pour alterner (tester l'indice, mettre un booléen alternativement à vrai et faux, ...).

4. ◇ En turtle, on peut spécifier une couleur de plusieurs façons. L'une de ces façons est d'utiliser un tableau de trois cases, chacune contenant une valeur entre 0 (couleur absente) et 1.0 (couleur maximum). La case 0 représente le rouge, la case 1 le vert et la case 2 le bleu (mode RVB ou RGB en anglais). On peut par exemple obtenir du jaune en mélangeant du vert et du rouge :

```
jaune=[1.0, 1.0, 0.0]
color(jaune)    #definit la couleur courante.
down()
goto(100, 100)   #trace un trait jaune.
```

Écrire un programme Python qui définit une distance et deux couleurs :

```
distance=200
couleur1=[1.0, 1.0, 0.0]    #du jaune
couleur2=[0.5, 0.1, 0.75]   #une espece de violet
speed(0)                    #vitesse maximale pour le curseur de dessin
up()
#Completer ci-dessous
```

puis qui dessine une bande horizontale dégradée de la couleur 1 à la couleur 2, de 100 pixels de haut. Cette bande sera constituée de traits verticaux de 1 pixel de large et de 100 pixels de hauts, dont la couleur doit varier *linéairement* entre la couleur 1 et la couleur 2. Plus précisément si la bande commence à $x = 0$ et se termine à $x =$ `distance`, alors :

— au point d'abscisse 0, le trait a la couleur 1
— au point d'abscisse `distance`, le trait a la couleur 2
— au point d'abscisse x la couleur y est donnée par une équation $y = a \times x + b$.
— les calculs se font composante par composante.

Réponse: Si on appelle la première couleur $c_1 = (r_1, v_1, b_1)$ et la deuxième couleur $c_2 = (r_2, v_2, b_2)$ et la distance d alors :

— $c_1 = 0 \times a + b$ donc $b = c_1$
— $c_2 = d \times a + b$ donc $a = \frac{c_2 - c_1}{d}$
Donc pour chaque point x, on calcule :

$$\begin{aligned} r &= \frac{r_2 - r_1}{d} \times x + r_1 \\ v &= \frac{v_2 - v_1}{d} \times x + v_1 \\ b &= \frac{b_2 - b_1}{d} \times x + b_1 \end{aligned}$$

En Python, les couleurs `couleur1` et `couleur2` étant dans un tableau, on peut calculer la couleur comme ceci :

```
1 for k in range(3):
2     couleur[k] = (couleur2[k] - couleur1[k])/distance * x + couleur1[k]
```