université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

vendredi 20 novembre 2025

Introduction à l'informatique
L1

TP n° 10

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.

But du TP

Le but de cette feuille d'exercice est :
— de se familiariser avec les types structurés en Python : n-uplets, dictionnaires et itérations sur ces derniers.
— de consolider les acquis sur les autres concepts, il y a donc une utilisation variée des tests, boucles, tableaux.

Exercices

Exercice 0

Ouvrir un terminal :
— créer un répertoire `IntroInfo` et un répertoire `TP7` à l'intérieur. Note : les fichiers des TPs précédents ne sont pas accessibles directement depuis cet environnement.
— se placer à l'intérieur du répertoire `TP7`
On suppose pour les autres exercices que le répertoire `TP7` est le répertoire courant.

Exercice 1

⋆

1. Pour chacun des programmes Python ci-dessous, dire ce qu'ils affichent.

(a)
```
1 def f (x, y):
2     return x+10, y-10
3
4 a, b = f(42, 43)
5 print("a =", a, ",b =", b)
```

(b)
```
1 def f (x):
2     if x % 2 == 0:
3         return 1
4
5 a = f(42)
6 b = f(43)
7 print("a =", a, ",b =", b)
```

(c)
```
1 def f(t):
2     t[0] = 42
3
4 t1 = [0, 0, 0]
5 t2 = [0, 0, 0]
6 f(t1)
7 print(t1)
8 print(t2)
```

(d)
```
1 def f(t):
2     t[0] = 42
3
4 t1 = [0, 0, 0]
5 t2 = t1
6 f(t1)
7 print(t1)
8 print(t2)
```

(e)
```
dic = { "A" : 1, "B" : 10, "C" : 3 }
for c in "XABCY":
   if c in dic:
       print (dic[c])
   else:
       print (0)
```

(f)
```
def f(d):
    d["A"] = 42
    d["B"] = 42
    d["C"] = d["C"] + 10
dic = { "A" : 3, "C" : 32, "D" : 10 }
f (dic)
print (dic)
```

Exercice 2

Le but de cet exercice est d'écrire une fonction permettant de trouver l'élément le plus fréquent d'un tableau d'entiers (*i.e.* qui apparait le plus de fois dans le tableau), en utilisant trois algorithmes différents.

On commence dans un premier temps par écrire une fonction permettant de générer des tableaux aléatoires d'entiers, puis on utilisera ces tableaux aléatoires pour tester les fonctions.

Dans cet exercice, vous devrez écrire une chaîne de documentation pour chacune des fonctions définies.

1. Écrire une fonction `tab_rand(l, a, b)` qui renvoie un tableau de taille `l` dont les valeurs sont comprises entre `a` et `b`. On pourra utiliser la fonction `randint(a, b)` du module `random`. Celle ci renvoie un entier aléatoire compris entre `a` et `b` inclus.

Même si les trois algorithmes que nous allons utiliser sont différents, ils ont en commun leur façon de se rappeler de la valeur la plus fréquente :

```
def algo(t):
  """trouve la valeur la plus fréquente dans le tableau t supposé non vide"""
   assert len(t) != 0
   max_ele = None
   max_occ = -1

   # Ici les algorithmes varient et examinent plusieurs candidats
   # du tableau t de façon différentes.
   # On appelle e l'élément que l'on considère et
   # occ_e le nombre d'occurrences de e trouvées jusqu'à présent
     if occ_e > max_occ:
     # s'il y a plus d'occurrences de e que l'élément le plus fréquent trouvé
     # jusqu'ici, on met à jour :
         max_ele = e
         max_occ = occ_e

   #...

  return (max_ele, max_occ)
```

Les trois fonctions demandées vont donc avoir cette même structure et différer dans la façon de parcourir le tableau `t`.

2. Le premier algorithme (naïf) consiste à considérer chaque élément e du tableau t avec une boucle (externe) puis à compter le nombre de fois où e apparaît dans le tableau avec une autre boucle (imbriquée). On peut ainsi déterminer facilement l'élément le plus fréquent. Programmer cette solution dans une fonction `algo1(t)` qui renvoie une paire `(e, n)` où `e` est l'élément le plus fréquent et `n` le nombre de fois où il apparait. En cas d'ex-aequo, vous pouvez renvoyer n'importe lequel de ces éléments. Votre fonction suppose que le tableau est non-vide.

3. Tester la fonction `algo1` sur des exemples écrits à la main, par exemple :

```
def test_algo1():
    assert algo1([2]) == (2, 1)
    assert algo1([2,3,7,7,1,1,7,9,10]) == (7, 3)
    assert algo1([10,10,2,3,7,7,1,1,7,9,10,10]) == (10, 4)
```

4. La fonction `time()` du module `time` renvoie le nombre de secondes écoulées depuis une date de référence. La valeur renvoyée est un nombre flottant, avec une précision supérieure à la miliseconde. Écrire une fonction `test()` qui
   — crée un tableau t de 5 cases contenant des tableaux aléatoires de taille `1`, `10`, `100`, `1000`, `10000` contenant des valeurs entre `0` et `100`.
   — pour chaque tableau $t[i]$, stocke le temps actuel (t_0), recherche l'élément le plus fréquent avec `algo1(t[i])`, stocke de nouveau le temps actuel (t_1) puis affiche le résultat de la recherche, la taille du tableau parcouru et le temps $(t_1 - t_0) \times 1000$ (le temps écoulé en millisecondes).
   Constater que l'algorithme est particulièrement mauvais (plusieurs secondes pour un petit tableau de 10000 éléments).
5. Le second algorithme, moins naïf, consiste à trier le tableau avant de le parcourir. En effet, un tableau trié a la propriété que tous les éléments égaux se suivent. Il suffit donc de parcourir une fois le tableau trié en se souvenant de l'élément précédent. Soit le tableau trié

```
      [ 50, 50, 50, 62, 62, 81, 81, 81, 81  ]
```

   On le parcourt dans l'ordre en maintenant un compteur du nombre d'éléments égaux vu jusqu'à présent. Lorsque l'on arrive sur le premier élément `62`, on voit qu'il est différent de son prédécesseur `50`. On sait donc qu'il y a trois `50`. On peut remettre notre compteur d'élément courant à `1` (pour compter le premier `62`). On prendra garde au cas particulier ou le tableau se termine par la répétition la plus fréquente, comme dans l'exemple ci-dessus. On rappelle que la fonction prédéfinie `sorted(t)` renvoie une copie triée du tableau `t`.
6. Mettre à jour la fonction `test` pour tester l'algorithme 2 dans un premier temps. Constater qu'il est plus rapide dès que le nombre d'éléments augmente. Puis, faire en sorte que la fonction teste dans le même tour de boucle le même tableau avec les algorithmes 1 et 2, pour vérifier que les valeurs renvoyées sont compatibles (i.e. soit identiques, soit qu'elles renvoie au moins le même nombre maximal d'occurences en cas d'éléments égaux). Constater que l'algorithme 2 est bien plus efficace que l'algorithme 1.
7. Le troisième algorithme consiste à utiliser un dictionnaire dans lequel on associe pour chaque entier rencontré un compteur lui correspondant.
   — initialiser un dictionnaire vide `dic`
   — pour chaque élément `e` du tableau, regarder si `dic[e]` est présent. Si ce n'est pas le cas, l'initialiser à `0`. Puis incrémenter cette valeur. Si elle est supérieure au nombre d'occurrence `max_occ` trouvé, alors `e` est le nouvel élément le plus fréquent.
8. Mettre à jour la fonction `test` pour tester l'algorithme 3 en même temps que le 2 et le 1. Comparer les temps relatif.
9. ◇ Commenter dans la fonction `test` les tests de l'algorithme 1, beaucoup trop naif. Tester avec de grandes tailles de tableau (`100 000` éléments et `1 000 000` d'éléments). Quel est l'algorithme le plus rapide ?

Exercice 3 ◇

On souhaite écrire un programme calculant des statistiques sur un fichier texte. On crée pour cela un fichier `stats.py`.

1. Écrire une fonction `stats (l)`. Cette dernière prend en argument un tableau de chaînes de caractères représentant les lignes d'un fichier (tel que renvoyé par `.readlines()` sur un descripteur de fichier). La fonction renvoie un dictionnaire de la forme `{ 'nombre de lignes': 42 }`, contenant le nombre de lignes dans le fichier.

2. Appeler la fonction dans le programme principal. Pour cela, ouvrir le fichier `/usr/share/common-licenses/GPL` [1] en lecture, renvoyer ses lignes comme un tableau que vous passerez à `stats`. Afficher pour chaque entrée du dictionnaire resultant la clé et la valeur correspondante (dans un premier temps, le programme n'affiche donc que le nombre de lignes). Tester votre programme sur le fichier
3. Modifier la fonction `stats` pour qu'elle sépare chaque ligne en liste de mots, puis déterminer le mot le plus long du fichier. Étant donné une chaîne s représentant une ligne, on peut obtenir les mots en deux étapes :
   — s`.split()` sépare les caractères selon les blancs (espaces, tabulation, retours à la ligne) et les renvoie un tableau t
   — pour chaque élément e de t, e`.strip(".!,:-;?")` retire les caractères de la chaîne `".!,:-;?"` se trouvant au début ou à la fin de e

   Ainsi :

```
1 s = "Hello, you! It's a beautiful day today, don't you think?\n"
2 t = s.split()
3 # t contient :
4 # ['Hello,', 'you !', "It's", 'a', 'beautiful', 'day',
5 # 'today,', "don't", 'you', 'think ?']
6
7 for e in t:
8     print(e.strip(".!,:-;?"))
```

   Le programme ci-dessus affiche :

```
Hello
you
It's
a
beautiful
day
today
don't
you
think
```

   Comme on le voit, les symboles de ponctuation se trouvant à l'extérieur d'un mot sont retirés. Les apostrophes se trouvant entre deux mots, sans espace, ne sont pas retirées. On considèrera alors que des chaînes telles que `"It's"` et `"don't"` constituent un même mot.

   La fonction `stat` rajoute au dictionnaire qu'elle renvoie une entrée « `"mot le plus long"`: m » associée au mot le plus long. Tester le programme.
4. Modifier la fonction `stats` pour qu'elle compte le nombre d'occurrences d'un mot et l'ajoute au dictionnaire renvoyé. Par exemple, si le mot `software` apparaît 14 fois dans le fichier testé, une entrée `"software": 14` doit se trouver dans le dictionnaire.
5. Modifier le code de la fonction `stat` précédente pour que les mots soient comptés indépendemment de leur casse ($i.e$ `Software`, `software`, `SOFTWARE` sont considérés comme le même mot). Étant donnée une chaîne w représentant un mot, on pourra utiliser w`.lower()` pour convertir ce mot en minuscules avant de le compter.
6. Importer la variable `argv` du module `sys`. Cette variable `argv` est un tableau contenant le nom du script en cours d'exécution ainsi que les arguments passés sur la ligne de commande. Par exemple, si on exécute un script Python comme ceci :

```
$ python3 stats.py abc def ghi
```

   Alors le tableau `argv` contiendra `['stats.py', 'abc', 'def', 'ghi']`. En d'autres termes, la case 0 du tableau contient le nom du script en cours d'exécution et les cases suivantes contiennent les arguments passés au script sur la ligne de commande. Votre programe doit vérifier que :

1. ce fichier contient la license décrivant les conditions d'utilisation d'une bonne partie des logiciels libres installés dans une distribution Linux

— il a été appelé avec exactement 1 argument (en plus du nom du script)
— cet argument est un fichier et qu'il peut être ouvert en lecture

Pour faire cette seconde vérification, on essayera d'ouvrir le nom de fichier correspondant et on rattrapera éventuellement l'exception levée par la fonction `open(...)`.

Attention : pour cette question vous ne pouvez plus tester votre programme directement depuis `geany`, mais il faudra le lancer depuis le terminal, en donnant explicitement le nom de fichier.