3 - Boucles for et Séquences (range, listes, chaînes de caractères)

Exercices et programmes à rendre dans le compte-rendu de TP :

ex 1 (TP3_1), ex 2 (TP3_2), ex 3 (TP3_3), ex 5 (TP3_5), ex 6 (TP3_6), ex 7 (TP3_7)

Ne pas rendre dans le compte rendu de TP : tableaux 1, 2, 3, 4, ex 4 (TP3_4)

3.1 - La boucle for

Remarque : les programmes `TP2_while` et `TP2_turtle` du TP précédent auraient pu (dû) être écrits avec une boucle `for`, car le nombre d'itérations était connu.

A partir de maintenant, nous utiliserons :
- une boucle `while` si le nombre d'itérations n'est pas connu à l'avance ;
- une boucle `for` si le nombre d'itérations est connu à l'avance.

Rappel :
`range(p,n,pas)` représente la suite des entiers de **`p` inclus** à **`n` exclu**, avec un **`pas`**.
On peut l'utiliser dans une boucle ou bien encore le convertir en liste.

Exercice 1 : Afficher des étoiles (programme TP3_1)

a) écrire un programme `TP3_1.py` qui permette de faire saisir un entier `nbEtoiles` par l'utilisateur et affiche `nbEtoiles` étoiles sur une même ligne en utilisant une boucle **`for`**, sans espace entre chaque étoile (le nombre d'étoiles sera représenté obligatoirement par la variable `nbEtoiles`). Il est interdit dans cet exercice d'utiliser la répétition de chaîne de caractère **`"*" * n`**.

*Indication : comme dans le programme `TP2_while`, utilisez l'option `end=` de la fonction **`print`**`()` pour que les étoiles s'écrivent sur une même ligne et non pas sur des lignes différentes.*

Exemple d'exécution (la saisie est en gras) :

```
combien d'étoiles ? 7
*******
```

b) modifier ce programme pour qu'il vérifie que le nombre d'étoiles est strictement positif et demande à l'utilisateur de le ressaisir en cas d'erreur.

Exemple d'exécution (les saisies sont en gras) :

```
Combien d'étoiles ? -2
Erreur, veuillez saisir un nombre strictement positif.
Combien d'étoiles ? 4
****
```

c) modifier votre programme pour qu'il permette de saisir un entier `nbLignes` strictement positif (contrôler cette saisie), et qu'il affiche `nbLignes` lignes de `nbEtoiles` étoiles en utilisant un second niveau de boucle **`for`** *(note : les noms des variables sont obligatoires).*

Exemple d'exécution (les saisies sont en gras) :

```
Combien d'étoiles ? 6
Combien de lignes ? 3
******
******
******
```

Exercice 2 : <u>somme d'entiers de 1 à n (programme TP3_2)</u>

On veut calculer la somme des entiers de 1 à n, sans utiliser la formule $\frac{n(n+1)}{2}$, mais à l'aide d'une boucle **`for`**.

a) Dans un premier temps, écrire une boucle **`for`** pour calculer la somme des entiers de 1 à 10. La somme sera stockée dans une variable nommée **`somme`**. Afficher cette somme à l'écran en fin de programme.

Exemple d'exécution :

```
La somme des entiers de 1 à 10 est 55.
```

b) Modifiez votre programme pour obliger l'utilisateur à saisir un nombre n strictement positif (vous pouvez copier-coller un morceau de votre programme `TP3_1.py`), puis calculez la somme des entiers de 1 à n, et affichez-la à l'écran en fin de programme.

Exercice 3 : <u>multiples de 7 (programme TP3_3)</u>

a) Écrire un programme qui affiche les 20 premiers multiples de 7 strictement positifs, séparés par un point-virgule sur une même ligne, mais en allant à la ligne après chaque multiple de 3.

b) Modifier ce programme pour qu'il :

- fasse saisir par l'utilisateur un nombre ***n*** et un nombre **`p`** strictement positifs ;

- affiche sur une même ligne les **`n`** premiers multiples de 7, séparés par un point virgule sur une même ligne, mais en allant à la ligne après chaque multiple de **`p`**.

c) Modifier ce programme pour que, en plus, il compte les multiples de **`p`** qui auront été affichés, et affiche ce nombre à la fin.

Exemple d'exécution :

```
Combien de multiples de 7 voulez vous afficher ? 20
Vous irez à la ligne après chaque multiple de ... (nbre stricte-
ment positif svp) : 3
7;14;21
28;35;42
49;56;63
70;77;84
91;98;105
112;119;126
133;140;
Vous avez affiché 6 multiples de 3
```

3.2 - Les listes

3.2.1 - Fonctions utiles pour travailler avec des listes

Les opérateurs, fonctions et méthodes les plus utiles pour les listes sont :

`len(lst)`, `lst.append()`, `lst.remove()`, `lst.pop()`, `elem in lst`, `lst.sort()`

Soit `lst` une liste quelconque, soit `elem` un élément :

`len(lst)`	renvoie le nombre d'éléments dans `lst`.
`lst.append(elem)`	ajoute l'élément `elem` à la fin de la liste `lst`.
`lst.remove(elem)`	supprime de la liste `lst`, la première occurence de l'élément `elem` (si la liste `lst` contient plusieurs fois l'élément `elem`, seul le premier est enlevé).
`lst.pop()`	supprime le dernier élément de la liste `lst`.
`lst.pop(i)`	supprime l'élément d'indice `i` de la liste `lst`.
`elem in lst`	renvoie `True` si `elem` est un élément de la liste `lst`, sinon renvoie `False`.
`lst.index(elem)`	renvoie l'indice de l'élément `elem` dans la liste `lst`.
`lst.sort()`	modifie la liste `lst` en la triant par ordre croissant.
`sorted(lst)`	crée une copie de la liste `lst`, triée par ordre croissant (la liste `lst` n'est pas modifiée par cette instruction ; seule sa copie est triée).

En tapant `help(list)` dans l'interpréteur python, on obtient toutes ces fonctions ainsi que beaucoup d'autres.

Nous allons maintenant apprendre à manipuler les listes. Saisissez dans l'interpréteur python :

Tableau 1 : manipuler les listes

instruction à saisir	résultat obtenu + commentaire ou explication
`maListe=[22,"coucou",33,"z",'a','b',111,99]`	
`maListe`	
`len(maListe)`	
`'z' in maListe`	
`maListe[3]=1024`	
`'z' in maListe`	
`maListe`	
`maListe.append(33)`	
`maListe.append("hello")`	
`maListe`	
`maListe.remove(33)`	
`33 in maListe`	
`maListe`	
`maListe.pop()`	

`maListe.pop(1)`	
`maListe`	
`maListe.index('a')`	
`maListe.index(111)`	
`maListe.sort()`	
`maListe.remove('a')`	
`maListe.remove('b')`	
`maListe`	
`maListe.sort()`	
`maListe`	
`uneListe=['z','a','d','aa']`	
`uneListe.sort()`	
`uneListe`	

On peut extraire une sous-liste d'une liste en indiquant entre crochets les indices des éléments que l'on veut extraire :

Extraction de sous-listes :

Soit **`lst`** une liste quelconque.

`lst[p]` renvoie l'élément d'indice `p` de **`lst`**.

`lst[p:n]` renvoie une nouvelle liste constituée des éléments de **`lst`** d'indice **`p`** inclus à **`n`** exclu.

`lst[p:n:pas]` renvoie une nouvelle liste constituée des éléments de **`lst`** d'indice **`p`** inclus à **`n`** exclu, tous les **`pas`**.

`lst[:]` renvoie une nouvelle liste constituée de tous les éléments de **`lst`**.

`lst[p:]` renvoie une nouvelle liste constituée de tous les éléments de **`lst`** à partir de l'élément d'indice **`p`** inclus.

`lst[:n]` renvoie une nouvelle liste constituée de tous les éléments de **`lst`** depuis le premier jusqu'à l'élément d'indice **`n`** exclu.

`lst[::pas]` renvoie une nouvelle liste constituée des éléments de **`lst`**, tous les **`pas`**.

Tableau 2 : extraire une sous-liste d'une liste

instruction à taper	résultat obtenu + commentaire ou explication
`maListe`	
`maListe[0:3]`	

`maListe[0:4:2]`	
`maListe[0:3]`	
`maListe[:]`	
`maListe[::2]`	
`maListe[::-1]`	
`maListe[0]`	
`maListe[1]`	
`maListe[4]`	
`maListe[5]`	
`maListe[-1]`	
`maListe[-2]`	
`maListe[-5]`	
`maListe[-6]`	
`maListe[:3]`	
`maListe[2:]`	
`maListe[-1]="b"`	
`maListe`	

L'affectation en Python est l'association entre un nom de variable et une valeur. Lorsqu'on affecte une variable à une autre variable, on crée simplement un autre nom qui partage la même valeur en mémoire - on peut tester qu'il s'agit effectivement de la même valeur avec l'opérateur **`is`** qui compare l'identité de deux valeurs. Essayez les instructions suivantes :

Tableau 3 : identité des valeurs

instruction à taper	résultat obtenu + commentaire ou explication
`a = 14000` [2]	
`b = a`	
`b`	
`b == a`	
`b is a`	
`c = 14000`	

2 On utilise une valeur entière élevée pour éviter un cas particulier d'optimisation réalisé pour les valeurs de -4 à 256, on aurait aussi pu utiliser une valeur flottante.

`c == a`	
`c is a`	

Cela ne pose pas de problème avec les types **`int`**, **`float`**, **`bool`** et **`str`** car les valeurs elle-mêmes ne sont pas modifiables[3] (pour changer la valeur associée à la variable il faut réaliser une nouvelle affectation).

De la même façon, l'affectation d'une variable à une autre variable d'une valeur de type **`list`**, ne crée pas une nouvelle liste qui soit une copie de la première, mais met simplement en place un nouveau nom qui référence la même liste en mémoire. Or il est possible de modifier directement cette valeur liste[4] en utilisant les opérateurs et méthodes vus précédemment. Si deux variables référencent la même liste, les modifications réalisées en utilisant une variable seront visibles aussi avec l'autre variable.

Pour créer une copie d'une liste en étant sûr que les modifications ne toucheront pas l'originale, il faut utiliser un des moyens présentés ci-après. Tapez dans l'interpréteur python :

Tableau 4 : copier une liste

instruction à taper	résultat obtenu + commentaire ou explication
`maListe`	
`liste2 = maListe`	
`liste2 is maListe`	
`liste2[1]="nouveau"`	
`liste2`	
`maListe`	
1ère méthode de copie : grâce à un transtypage. `liste3 = list(maListe)`	
`liste3 is maListe`	
`liste3[4]="surprise"`	
`liste3`	
`maListe`	
2ème méthode de copie : grâce à une extraction de tous les éléments de maListe. `liste4 = maListe[:]`	
`liste4 is maListe`	
`liste4[0]=8`	
`liste4`	

3 En Python en dit que ces types int, float, bool, str sont "immutables".
4 En Python on dit que le type list est "mutable".

`maListe`	

3ème méthode de copie : avec le module copy
`import copy`
`liste5 = copy.deepcopy(maListe)`	

`liste5 is maListe`	

La troisième méthode a l'avantage de faire une copie en profondeur sur tous les éléments de la liste, ce qui permet de traiter le cas des listes imbriquées… On peut en effet mettre des listes dans des listes, par exemple pour créer des tableaux à 2 dimensions (ou plus) :

Tableau 5 : liste de listes

instruction à taper	résultat obtenu + commentaire ou explication
`var=[[1,2,3],[4,5,6]]`	
`var[0]`	
`var[1]`	
`var[0][0]`	
`var[0][1]`	
`var[0][2]`	

Tout comme pour les chaînes de caractères, on peut concaténer deux listes avec l'opérateur `+` et répliquer plusieurs fois le contenu d'une liste avec l'opérateur de multiplication `*` et un entier .

3.2.2 - Parcourir ou remplir une liste avec une boucle for

On peut parcourir une liste avec une boucle `for` de 2 façons :

- soit en parcourant les indices de la liste :

```
for indice in range(len(lst)):
```

- soit en parcourant directement la liste :

```
for element in lst:
```

Exercice 4 : utiliser une liste avec des boucles for (TP3_4)

a) Ouvrez le programme `TP3_4.py`. Il contient uniquement la déclaration et l'initialisation de la variable suivante, de type list :

```
semaine=["lundi","mardi","mercredi","jeudi","vendredi","samedi",
         "dimanche"]
```

Écrivez une boucle `for` pour afficher à l'écran les éléments de la liste **`semaine`**, sur des lignes différentes, avec la première méthode (parcours des indices), puis avec la deuxième méthode (parcours direct de la liste).

b) Dans ce même programme, on veut créer et remplir une liste nommée **`calendOct11`**, qui contiendra toutes les dates complètes des 31 jours du mois d'octobre 2011 qui a commencé un

samedi. Dans le programme, une instruction de création d'une liste vide `calendOct11` est déjà écrite :

```
calendOct11 = []
```

Écrivez une boucle pour remplir cette liste `calendOct11`, afin d'obtenir la liste suivante :

```
['samedi 1 octobre', 'dimanche 2 octobre', ...(etc)]
```

A la fin du programme, cette liste sera affichée à l'écran grâce à l'instruction :

```
print(calendOct11)
```

Exercice 5 : produit scalaire

On représente les vecteurs en dimension 3 et en repère orthonormé par des listes de 3 nombres flottants.

Ecrivez un programme `TP3_5` qui calcule et affiche le produit scalaire de 2 vecteurs `u` et `v`.

Les vecteurs `u` et `v` ne seront pas saisis par l'utilisateur mais seront écrits directement dans le programme. Par exemple vous pourrez prendre tout d'abord `u=[1.0,2.0,3.0]` et `v=[4.0,5.0,6.0]`. Puis, pour tester votre programme sur d'autres vecteurs, vous changerez les valeurs de `u` et `v` directement dans votre programme.

Calculez également le cosinus de l'angle entre `u` et `v`, et affichez-le.

Mémo, produit scalaire : $\vec{u}\cdot\vec{v}=u_x\times v_x+u_y\times v_y+u_z\times v_z$

Mémo cosinus entre les angles : $\cos(\vec{u};\vec{v})=\frac{\vec{u}\cdot\vec{v}}{|\vec{u}|\times|\vec{v}|}$

3.3 - Les chaînes de caractères

3.3.1 - Fonctions utiles pour travailler avec des chaînes

Les chaînes de caractères fonctionnent à peu près comme des listes de caractères, mais elles ne sont pas modifiables : on ne peut donc pas leur ajouter des caractères ou en enlever, ni trier leurs caractères par ordre croissant, ni modifier les caractères un par un.

En revanche, on peut accéder aux caractères d'une chaîne par leurs indices dans la chaîne, et extraire des sous-chaînes d'une chaîne de la même façon que pour les listes.

`uneChaine[p:n:pas]` — Renvoie une nouvelle chaîne constituée des caractères de `uneChaine` d'indice `p` inclus à `n` exclu, tous les `pas`.
`unCar in uneChaine` — Renvoie `True` si le caractère `unCar` est présent dans la chaîne `uneChaine`, sinon renvoie `False`.
`len(uneChaine)` — Renvoie la longueur (le nombre de caractères) de la chaîne `uneChaine`

Certains caractères se traduisent par un retour à la ligne ou une tabulation.

le caractère `\n` introduit un retour à la ligne
le caractère `\t` introduit une tabulation (décalage horizontal)
le caractère `\\` introduit un seul `\`

Tableau 6 : manipuler les chaînes de caractères

instruction à taper	résultat obtenu + commentaire ou explication
`s1="bonjour"`	
`len(s1)`	
`s1[:3]`	
`s1[2]="z"`	
`o in s1`	
`print("vive\n l'informa\ntique")`	
`"ara"=="zygomatique"`	
`"ara"<"zygomatique"`	
`"ara">"zygomatique"`	
`"abscisse"<"zébu"`	
`"abscisse"=="zébu"`	
`"abscisse">"zébu"`	

Exercice 6 : compter des voyelles (TP3_6)

Ecrire un programme qui fait saisir une phrase par l'utilisateur, puis qui compte le nombre de voyelles dans cette phrase, et enfin qui affiche ce nombre à l'écran.

Exercice 7 : compter des mots (TP3_7)

Ecrire un programme qui effectuera les tâches suivantes :

- Faire saisir une phrase avec input()

- Compter les mots de cette phrase, puis afficher ce nombre de mots.

- on considérera que le séparateur des mots est le caractere 'espace'

a) Tout d'abord, on considérera que l'utilisateur commence forcément sa phrase par une lettre, qu'il ne met pas plus de 1 espace entre chaque mot, et qu'il termine sa phrase par un caractère qui n'est pas un espace.

b) Facultatif : maintenant, l'utilisateur peut entrer une phrase commençant par un ou plusieurs espaces, il peut séparer les mots de la phrase par plus d'un espace et il peut terminer sa phrase par des espaces. (indication : utilisez une variable booléenne `mot_en_cours` pour indiquer si on est déjà dans un mot au moment où on rencontre un caractère.)

3.3.2 - Autres fonctions concernant les chaînes de caractères

D'autres méthodes utiles pour les chaînes de caractère (certaines acceptent d'autres paramètres optionnels, qui ne sont pas listés ici - la notation *[xxx]* indique que xxx est optionnel et peut être omis - les *[]* ne sont qu'une indication et ne font pas partie de la syntaxe lors de l'utilisation) :

uneChaine.split(*[séparateur]*) — Renvoie une liste contenant la chaîne **uneChaine** découpée en plusieurs sous-chaînes. Par défaut la séparation se fait sur les blancs (espaces, tabulations, retours à la ligne), sauf si un autre **séparateur** est spécifié.

uneChaine.join(uneListeDeChaines) — Renvoie une nouvelle chaîne contenant tous les éléments de la liste **uneListeDeChaines**, concaténés en utilisant **uneChaine** comme séparateur.

uneChaine.find(sousChaîne) — Renvoie la position de la première occurrence de **sousChaîne** dans **uneChaine**. Renvoie **-1** si **souChaîne** n'est pas trouvée.

uneChaine.strip(*[caractères]*) — Renvoie une nouvelle chaîne dans laquelle tous les blancs (ou tout caractère présent dans **caractères** s'il est donné en paramètre) sont ôtés au début et à la fin de **uneChaine**.

uneChaine.replace(ancienne, nouvelle) — Renvoie une nouvelle chaîne dans laquelle chaque occurrence dans **uneChaine** de la sous-chaîne **ancienne** est remplacée par **nouvelle**.

uneChaine.capitalize() — Renvoie une nouvelle chaîne dans laquelle le premier caractère de **uneChaine** a été transformé en majuscule et les suivants en minuscules.

uneChaine.lower() — Renvoie une nouvelle chaîne où toutes les lettres de **uneChaine** ont été converties en minuscules.

uneChaine.upper() — Renvoie une nouvelle chaîne où toutes les lettres de **uneChaine** ont été converties en majuscules.

3.4 - Pour les rapides

Supplément 1 : Adaptez votre programme de l'exercice 1 (TP3_1) pour qu'il permette de saisir un caractère (qui sera stocké dans une variable nommée `unCar`), et qu'il affiche p lignes de n caractères `unCar`.

Supplément 2 : Modifiez votre programme de l'exercice 2 (TP3_2) pour qu'il affiche tous les nombres de la somme séparés par un + , suivis de = et de la somme résultat.

Exemple d'exécution :

```
Entrez un nombre strictement positif: 11
1+2+3+4+5+6+7+8+9+10+11=66
```

3.5 - Travail personnel

Exercice 8 : combinaisons de dés (TP3_8)

On joue à lancer un dé rouge à 6 faces et un dé vert à 6 faces. On se demande combien de façons il y de faire un nombre n en additionnant la face du dé rouge et celle du dé vert. (par exemple, il y a une seule façon d'obtenir 2 qui est de faire 1 avec le dé vert et 1 avec le dé rouge. Il y a 2 façons d'obtenir 3, qui sont de faire 1 avec le vert et 2 avec le rouge, ou bien 1 avec le rouge et 2 avec le vert).

- Demander à l'utilisateur de saisir une valeur entière.

- A l'aide de boucles imbriquées, compter le nombre de possibilités pour obtenir ce nombre comme somme du dé rouge et du dé vert.

- Affichez le résultat.

Exemple d'exécution:

```
affichage : entrez un nombre entier :
saisie : 9
affichage : il y a 4 façon(s) de faire 9 avec deux dés.
```

Exercice 9 : caractériser des triangles (TP3_9)

Demander à l'utilisateur de saisir trois longueurs a, b, c. Déterminer s'il est possible de construire un triangle de côtés a, b et c. Si oui, déterminer si ce triangle est rectangle, isocèle, équilatéral ou quelconque. (remarque : un triangle rectangle peut être également isocèle). Enfin, afficher les résultats.

Exercice 10 : les diviseurs d'un nombre (TP3_10)

Ecrire un programme qui demande à l'utilisateur de saisir un nombre entier strictement positif, et qui affiche la liste de ses diviseurs. A la fin du programme, indiquer également le nombre de diviseurs de cet entier.