TP 1 : Boucles et listes

Objectifs du TP. Écrire correctement des boucles `for` et `while`, appendre à manipuler des listes.

1 Rappels du TP précédent

• Pour $m < n$ deux entiers, `range(m,n)` produit un *itérable*, fournissant la suite des entiers de m à $n-1$ (on peut écrire `range(n)` à la place de `range(0,n)`). On fait parcourir à une variable `i` ces entiers-là à l'aide d'une boucle `for` ayant la forme suivante :

```
for i in range(m,n):
    [instructions]
```

```
s=0
for i in range(1,10):
    s=s+i #s contiendra la somme des entiers de 1 à 9
```

Les instructions indentées étant répétées pour tout i entre m et $n-1$. On peut de plus spécifier un pas p : pour $p > 0$, `range(m,n,p)` permet d'obtenir les entiers $m, m+p, m+2p...$ strictement inférieurs à n. Pour $p < 0$, `range(m,n,p)` fournit les entiers $m, m-p, m-2p...$ strictement supérieurs à n. Par exemple `range(m,-1,-1)` fournit les entiers de m à 0.

- On rappelle le fonctionnement d'une boucle `while` : tant qu'une certaine condition est vérifiée, on réalise les instructions de la boucle. Plus précisément, à chaque fois que l'expression booléenne qui suit le `while` s'évalue en `True`, on fait un tour de boucle et on réévalue la condition. Dès qu'elle s'évalue en `False`, on passe aux instructions situées après la boucle.

```
while expression:
    [instructions]
```

```
a=123
while a>0:
    print(a) #on affiche à l'écran les quotients dans les
    a=a//2 #divisions successives de 123 par 2
```

Exercice 0. *Variante du TP précédent.* ***1.*** Calculer $\sum_{k=0}^{100} \frac{(-1)^k}{k!}$ à l'aide d'une boucle `for`. Vérifier que cette somme vaut environ $1/\mathrm{e}$. Dans le module `math` se trouvent la fonction factorielle (`factorial`) et la constante `e`.

2. On vous donne deux entiers $N > 0$ et $b \geq 2$. À l'aide d'une boucle `while` et de divisions euclidiennes, écrire un code permettant d'obtenir le nombre de chiffres de N dans la base b. (Rappel : `//` pour effectuer une division euclidienne).

2 Les listes

Une liste est la donnée d'une séquence finie d'éléments, possiblement de types différents. Voici un exemple de liste : `[True, 4, 5, 3.0]`. En pratique, on ne manipulera que des listes homogènes, c'est-à-dire constituées d'éléments de même type.

2.1 Accès aux éléments d'une liste. Modification.

Accès aux éléments. Pour `L` une liste, sa *longueur* (nombre d'éléments, length en anglais) est accessible avec `len(L)`. En notant n cette longueur, les éléments sont indexés par les entiers de 0 à $n-1$. Exemples :

```
>>> L = [1, 2, 3, 4, 5]
>>> L[2]
3
>>> L[len(L)-1]
5
```

Ainsi, `for i in range(len(L)):` est le début d'une boucle permettant à la variable i de parcourir les indices de tous les éléments de la liste.
Si on demande l'accès à un caractère d'indice négatif i compris entre -1 et $-n$, où n est la longueur de la liste, celui-ci est considéré comme étant $n+i$:

```
>>> L[-1] # très pratique pour accéder au dernier élément !
5
>>> L[-5]
1
```

L'accès à tout autre indice produit une erreur :

```
>>> L[len(L)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Retenez bien cette erreur, vous l'aurez souvent !

Exercice 1. *Parcours de listes.* On travaille uniquement sur des listes d'entiers. On suppose la variable `L` affectée, et que celle-ci contient une liste de nombres non vide. Vous pouvez chosir n'importe quoi, comme par exemple la liste `L=[0, -1, 3, 5, 9, 11, 2, 1, 5]`.

1. Écrire une suite d'instructions permettant d'affecter à la variable `somme_liste` la somme des éléments de la liste.
2. Écrire une suite d'instructions permettant d'affecter à la variable `max_liste` l'élément maximal se trouvant dans la liste.
3. Écrire une suite d'instructions permettant d'affecter à la variable `indice_max_liste` l'indice de l'élément maximal se trouvant dans la liste (c'est-à-dire l'élément i tel que `L[i]` est maximal. Si l'élément maximal se trouve à plusieurs endroits, `indice_max_liste` devra contenir le premier indice où ce maximum est rencontré).
4. On dit qu'une liste `L` de longueur n est croissante si pour tout i entre 0 et $n-2$, on a $L[i+1] \geq L[i]$. Écrire une suite d'instructions permettant d'affecter à la variable `est_croissante` un booléen (`True` ou `False`), ce booléen valant `True` si et seulement si la liste est croissante.

Testez vos codes, y compris avec d'autres listes !

Modification d'un élément. Étant donnée une liste `L`, on peut modifier l'élément d'indice i de `L` en le remplaçant par l'élément de son choix. La syntaxe est la même que pour une affectation.

```
>>> L=list(range(1,6))
>>> L[2]=10
>>> L
[1, 2, 10, 4, 5]
```

Les règles régissant l'indice i sont les mêmes que précédemment.

2.2 Construction de listes

Possibilités de construction. On peut construire une liste de plusieurs manières :

- par la donnée explicite des éléments, entre crochets, séparés par des virgules, comme dans `[1, 2, 3]`.
- par concaténation de listes (à l'aide de `+`) : `[1, 2, 3]+[4, 5, 6]` s'évalue en `[1, 2, 3, 4, 5, 6]`. Pour construire directement une liste de p éléments contenant le même élément (par exemple 0), on peut concaténer p fois la liste `[0]` avec elle-même avec l'instruction `[0]*p`.
- `list(iterable)` permet de fabriquer une liste à partir d'un itérable. Par exemple, `list(range(4))` s'évalue en `[0, 1, 2, 3]` et `list("truc")` en `['t', 'r', 'u', 'c']`.
- par compréhension, très pratique avec la structure suivante : `L=[f(x) for x in iterable if P(x)]`, où `f(x)` est une expression dépendant (ou non) de `x`, et `P(x)` est une expression booléenne (facultative). Par exemple :
  - `[x*x for x in range(5)]` s'évalue en la liste `[0, 1, 4, 9, 16]`.
  - `[x*x for x in range(5) if x%2==0]` s'évalue en la liste `[0, 4, 16]`.
- par *slicing* (tranchage), qu'on ne verra pas dans ce TP.
- par ajouts d'éléments successifs avec `append`, en partant d'une liste vide `[]`. Voir la suite !

Rajout d'un élément à une liste : la méthode `append`. Parmi les nombreuses *méthodes* qui existent sur les listes, l'une d'entre elles nous servira très souvent : `append`. Elle permet de rajouter un élément à la fin d'une liste :

```
>>> L=[1, 2, 3]
>>> L.append(8)
>>> print(L)
[1, 2, 3, 8]
```

Il est très fréquent de créer une liste en partant de la liste vide `[]`, qu'on remplit avec une boucle `for` et `append`.

Exercice 2. *Création de listes.*

1. De deux manières différentes, créer une liste `L0` de taille 100 ne contenant que des zéros ;
2. De même, créer la liste `L100` contenant tous les entiers entre 0 et 99, dans cet ordre, de plusieurs manières :
   - avec `list` ;
   - avec une boucle `for` et `append`.
3. Créer la liste `L300` contenant tous les entiers de 0 à 99, trois fois : le début est `[0, 0, 0, 1, 1, 1, 2,...]`. On utilisera deux boucles `for` imbriquées et `append`.
4. Créer une liste qui contient tous les entiers de 0 à 99, puis ceux de 98 à 0 (elle a donc 199 éléments).
5. Créer par compréhension la liste des puissances de 2, de 0 à 20.
6. Créer par compréhension la liste des entiers inférieurs à 100 divisibles par 3 ou par 5, mais pas par 15.

3 Exercices

3.1 Les algorithmes du cours pour les entiers naturels

Exercice 3. *De la base 10 à une base quelconque.* Écrire en Python l'algorithme du cours, permettant de calculer les chiffres d'un entier N dans la base b par divisions successives. On rappelle que `//` permet d'obtenir le quotient dans une division euclidienne, et `%` le reste. Avec $N = 158$ et $b = 7$, vous devez obtenir `[4, 1, 3]`, alors qu'en base 2 vous devez avoir `[0, 1, 1, 1, 1, 0, 0, 1]` : les chiffres sont stockés du poids faible au poids fort.

Exercice 4. Inversement, écrire un code permettant de passer d'une liste `L` représentant un entier dans la base b à son interprétation en base 10. Testez-le !

3.2 Encore un exercice sur les listes

Exercice 5. Construire pour l'exemple une liste `L` contenant uniquement des entiers (pas forcément tous positifs, comme `L=[0, -1, 3, 5, -9, 11, 2, 1, 5]`).

1. À l'aide d'une boucle `for`, écrire une séquence d'instructions affichant à l'écran le nombre de zéros et de uns. On utilisera `print("le nombre de zéros est : ", nb_zeros)` pour l'affichage, avec `nb_zeros` une variable contenant le nombre de zéros, de même pour le nombre de uns. Modifier `L` si besoin pour vérifier.
2. L'expression Python `x in L` s'évalue un booléen (`True` ou `False`) indiquant si `x` est dans `L`. Écrire un morceau de code à l'aide d'une boucle `while` permettant de calculer le plus petit entier *naturel* non présent dans `L`. Modifier `L` si besoin pour vérifier.

3.3 Pour aller plus loin

Exercice 6. *Une reprise du TP0.* Écrire un code permettant de tester si un entier p stocké dans la variable du même nom est premier : il suffit de tester la divisibilité de p par tous les entiers entre 2 et $p - 1$. Affecter le résultat (`True` ou `False` dans une variable booléenne `b`). Tester avec 65521 (premier) et 96709 (non premier).

Exercice 7. La méthode du crible d'Eratosthène permet de calculer efficacement tous les entiers premiers strictement inférieurs à un entier $N > 2$. Pour ce faire :

- on crée une liste `L` de booléens de taille N, initialement tous `True`.
- on « décoche `L[0]` et `L[1]` (c'est-à-dire qu'on affecte `False` à ces positions dans la liste) : ils ne sont pas premiers.

— on effectue ensuite une boucle sur les indices entre 2 et $N-1$:
  — `L[2]` est `True`, donc 2 est premier. On peut « décocher » tous les multiples non triviaux de 2 (4, 6, 8, etc...) : ils ne sont pas premiers.
  — `L[3]` est `True`, donc 3 est premier. On peut « décocher » tous les multiples non triviaux de 3 (6, 9, 12, etc...) : ils ne sont pas premiers.
  — `L[4]` est `False`, donc 4 n'est pas premier.
  — `L[5]` est `True`, donc 5 est premier. On peut « décocher » tous les multiples non triviaux de 5 (10, 15, 20, etc...) : ils ne sont pas premiers.
  — etc...

À l'aide de cette méthode, construire la liste de tous les entiers premiers inférieurs à 10^3 (vous devez en trouver 168). Combien y a-t-il d'entiers premiers inférieurs à 10^6 ?

Exercice 8. Cet exercice est plus délicat. Une sous-séquence croissante d'une liste `L` est une portion d'éléments d'éléments contigus, qui sont dans l'ordre croissant. Par exemple dans la liste `[0, 2, 8, 1, 5, 7, 9, 11, 2, 4, 6, 0, 1]`, la portion `[0, 2, 8]` est une sous-séquence croissante. Écrire un morceau de code permettant de calculer la taille d'une sous-séquence croissante maximale (dans cet exemple, il s'agit de `[1, 5, 7, 9, 11]`, de taille 5).