TP 5
Listes

Les listes constituent le dernier concept absolument fondamental du langage Python que nous étudions dans ce début d'année. Le lien avec les boucles `for`, ainsi qu'avec les calculs de suites et de sommes, apparaitra rapidement. Puis les listes permettront de poser de nombreuses questions algorithmiques nouvelles.

I Notion de liste

Une liste sert à contenir plusieurs objets, rangés les uns à la suite des autres. Ils sont écrits entre crochets et séparés par des virgules. Les listes sont des types de variables à part entière, qu'on peut mettre dans les variables ou donner en argument à des fonctions. Voici par exemple une liste de cinq nombres impairs :

```
L = [1, 3, 5, 7, 9]
```

Mais les éléments de la liste peuvent être de type quelconque, et même pas forcément tous du même type ! Voici par exemple une liste de courses :

```
courses = ["oeufs", "pain", "riz", "beurre"]
```

I.1 Accès aux éléments

Les éléments d'une liste sont numérotés à partir de 0. À tout moment on peut accéder au i-ème élément de la liste `L` avec la syntaxe `L[i]` et le modifier. Avec les listes ci-dessus :

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> courses[0]
'oeufs'
>>> courses[2]
'riz'
>>> courses[2] = "pâtes"
>>> courses
['oeufs', 'pain', 'pâtes', 'beurre']
```

La fonction `len(L)` donne la longueur de la liste `L`. **Dans une liste de longueur n, les éléments sont numérotés de 0 à $n-1$ inclus**. En mathématiques on noterait par exemple $(x_0, x_1, ..., x_{n-1})$ une telle liste. Ce sont les même conventions que pour `range(n)`, et ce sera bien pratique.

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> len(courses)
4
>>> L = [1, 3, 5, 7, 9]
>>> len(L)
5
```

Les indices négatifs correspondent à parcourir la liste en sens inverse.

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> course[-1]
'beurre'
>>> course[-2]
'riz'
```

Si les indices dépassent la longueur de la liste, on obtient une erreur.

```
>>> courses = ["oeufs", "pain", "riz", "beurre"]
>>> course[4]
IndexError: list index out of range
```

Cette erreur est extrêmement fréquente : la liste est de longueur 4, donc le dernier élément est `courses[3]`, il n'y a pas d'élément d'indice au-delà de 4…

Enfin la **liste vide** est notée tout simplement `[]` et est de longueur 0. Elle sera loin d'être inutile.

Exercice 1
Créez une liste `repas` contenant votre dernier repas, et testez vous-même les syntaxes précédentes.

I.2 Opérations sur les listes

L'opération `+` entre listes s'appelle la **concaténation**. La liste `L + M` est composée de la liste `L` à laquelle est mise bout à bout la liste `M`.

```
>>> ["oeufs", "pain", "riz", "beurre"] + ["fromage", "pommes"]
['oeufs', 'pain', 'riz', 'beurre', 'fromage', 'pommes']
```

Exercice 2
Ajoutez à votre liste `repas` de l'exercice 1 une nouvelle liste contenant le repas dont vous rêvez.

Pour un nombre entier `n` il y a aussi une opération de multiplication `*` entre une liste et `n` : le résultat `L * n` est la même chose que `L + L + ... + L` (`n` fois).

```
>>> ["oui", "non"] * 3
['oui', 'non', 'oui', 'non', 'oui', 'non']
```

Exercice 3
Quelle est la syntaxe la plus simple pour créer une liste de n zéros ?

Nous avons déjà vu ces opérations sur les chaines de caractères. Revoir le TP 1 : l'opération `+` « colle » deux chaines de caractères

```
>>> "BCPST" + "1B"
'BCPST1B'
```

et l'accès au i-ème caractère suit les mêmes règles, si `s = "BCPST"` alors `len(s)` est 5, `s[0]` est le caractère seul `B` et `s[4]` est le `T`, qui est aussi `s[-1]` ici.

Une opération encore plus courante consiste à ajouter un élément `x` seul à la fin de la liste `L`. Pour cela la syntaxe est `L.append(x)`. Cette opération ne renvoie pas de valeur (en mode interactif, rien ne s'affichera), mais modifie la liste elle-même :

```
>>> L = ["oeufs", "pain", "riz", "beurre"]
>>> L.append("fromage")
>>> L
['oeufs', 'pain', 'riz', 'beurre', 'fromage']
```

Exercice 4
Dans votre liste `repas`, ajoutez un légume puis un fruit, c'est important pour la santé !!!

L'opération « inverse » est `L.pop()` qui supprime le dernier élément de la liste **et le renvoie**. Ainsi on peut récupérer l'élément avec `x = L.pop()` pendant qu'il est supprimé de `L`. Avec la même liste précédente :

```
>>> L
['oeufs', 'pain', 'riz', 'beurre', 'fromage']
>>> x = L.pop()
>>> L
['oeufs', 'pain', 'riz', 'beurre']
>>> x
'fromage'
```

Remarque. Qu'est-ce que c'est que cette syntaxe ? Et pourquoi pas une syntaxe telle que `append(L, x)` ou bien `L = append(L, x)` ?

C'est la première fois que nous la rencontrons vraiment. Disons en première approche que tout se passe comme si *chaque* liste `L` venait avec *sa* propre fonction `append()` qui peut modifier la liste ; pour une autre liste `M` ce sera `M.append(x)`. Remarquez que les autres opérations vues jusque là ne modifiaient pas la liste `L`, alors que **`append` et `pop` modifient la liste elle-même**. On parle de **méthodes** plutôt que de fonctions. Dans l'aide interactive `help(list)` on trouve toutes les méthodes de base applicables sur les listes.

Une autre possibilité, qui donne en apparence exactement le même résultat final, est d'écrire

```
L = L + [x]
```

Cependant cela oblige à :
- Créer une liste `[x]` ne contenant qu'un seul élément `x`,
- Concaténer cette liste au bout de `L`,
- Modifier la variable `L` pour que cela devienne ce `L + [x]` qu'on vient de former,

Cela constitue en fait beaucoup plus d'opérations pour l'ordinateur.

À retenir

`append` et `pop` modifient la liste elle-même, alors que `+` et `*` ne sont que des opérations comme les autres, dont le résultat est une nouvelle liste et peut ensuite être mis dans une variable.

Remarque. Quand on les manipule uniquement avec `append` et `pop`, les listes se comportent comme des **piles** — exactement comme une pile de livres sur son bureau. Les opérations de base consistent à poser un élément sur le haut de la pile, ou à récupérer l'élément du dessus. On récupère ainsi les éléments selon l'ordre inverse duquel on les a ajoutés. Si on applique `pop` sur une liste vide, on obtient une erreur `IndexError: pop from empty list` car il n'y a plus rien à enlever.

II Méthode : itérer sur les listes

En pratique, si on reçoit une liste quelconque `L`, on ne sait pas forcément à l'avance quels sont ses éléments. Il est nécessaire d'utiliser une boucle pour « effectuer une opération » sur chaque élément de la liste. On parle de **parcourir la liste**.

La méthode de base s'appelle **itérer sur les indices**. Une liste de longueur n est numérotée de 0 à $n-1$ et donc une boucle `for i in range(n)` va parcourir tous les indices de la liste, permettant d'accéder à `L[i]` et de faire une opération dessus. Remarquez que la convention de numérotation des listes est bien compatible avec celle de `range`. Exemple :

```
L = ["oeufs", "pain", "riz", "beurre"]
for i in range(len(L)):
    print("indice :", i, "élément :", L[i])
```

produit le résultat

```
indice : 0 élément : oeufs
indice : 1 élément : pain
indice : 2 élément : riz
indice : 3 élément : beurre
```

C'est intéressant quand ces morceaux de programme font partie d'une fonction. Donnons l'exemple suivant qui lit une liste de nombres et affiche le double de chacun.

```
def double(L):
    for i in range(len(L)):
        print(2 * L[i])
```

Essai :

```
>>> L = [1, 3, 5]
>>> double(L)
2
6
10
```

Exercice 5
Écrire une fonction `signe(L)` qui prend en argument une liste `L` de nombres et qui affiche pour chacun des éléments le mot `"positif"`, `"négatif"` ou `"nul"`, selon son signe. On doit par exemple avoir :

```
>>> signe([4, 0, 7, -5])
positif
nul
positif
négatif
```

III Méthode : créer une liste à partir de zéros

Quand on sait à l'avance qu'on veut une liste de n nombres, une méthode intéressante est de commencer par créer une liste de n zéros avec la syntaxe `L = [0] * n`, puis de remplir la liste au fur et à mesure. C'est particulièrement intéressant pour les suites dont chaque terme dépend du ou des précédents.

Un exemple de base est la fonction qui crée la liste des n premières puissances de 2 :

```
def puissances2(n):
    L = [0] * n
    L[0] = 1
    for i in range(1, n):
        L[i] = 2 * L[i-1]
    return L
```

Remarquez comme la formulation ressemble à la définition d'une suite $(u_i)_{0\leqslant i<n}$ avec $u_0 = 1$ et $\forall 1 \leqslant i < n$, $u_i = 2 \times u_{i-1}$. Remarquez bien les bornes de la boucle `for`, puisqu'on veut écrire `L[i-1]` il faut partir de `i` à 1 et pas 0… L'exécution donne bien le résultat voulu, les puissances de 2^0 à 2^4 :

```
>>> puissance2(5)
[1, 2, 4, 8, 16]
```

Exercice 6
Écrire une fonction `factoriel(n)` *(encore ?)*, qui renvoie la liste des valeurs $i!$ pour $0 \leqslant i \leqslant n$, où $i! = 1 \times 2 \times \cdots \times i$, et pour $i = 0$ c'est 1.

Le cas des suites récurrentes d'ordre 2 n'est pas du tout plus difficile : chaque terme de la suite dépend des deux termes précédents, mais précisément la liste sert à se souvenir de *tous* les termes précédents, donc il n'y a pas besoin d'astuce particulière comme au TP précédent.

Exercice 7
Écrire une fonction `fibonacci(n)` *(encore ? et ce n'est que le début)* qui renvoie la liste des n premiers termes de la suite de Fibonacci $(F_i)_{i\in\mathbb{N}}$ avec $F_0 = 0$, $F_1 = 1$ et $\forall i \geqslant 2$, $F_i = F_{i-1} + F_{i-2}$.

IV Méthode : créer une liste par append successifs

Une autre méthode courante pour créer des listes est de démarrer avec une liste vide `L = []` et d'utiliser des appels à `L.append()` pour ajouter des éléments uns par uns. Cela est intéressant notamment quand on ne connait pas à l'avance la taille de la liste finale, on étudie un certain problème ou une certaine équation et on rajoute des solutions qu'on trouve au fur et à mesure.

Comme exemple de base, donnons une fonction qui prend en argument une liste `L` et renvoie une liste `P` constituée uniquement des termes de `L` qui sont positifs (donc ici c'est `P` qui grandit donc on utilise `P.append()`).

```
def garde_positifs(L):
    P = []
    for i in range(len(L)):
        if L[i] >= 0:
            P.append(L[i])
    return P
```

Test :

```
>>> garde_positifs([4, -8, -2, -5, 0, 1, 1, 6, 6, -2, 2, 1])
[4, 0, 1, 1, 6, 6, 2, 1]
```

Exercice 8 *Équation de Pell-Fermat*
On recherche des solutions entières à l'équation suivante :

$$x^2 - 3y^2 = 1, \quad (x, y) \in \mathbb{Z}^2$$

Comme il pourrait y avoir une infinité de solutions (il se peut très bien que x et y soient tous les deux extrêmement grands et que pourtant la différence $x^2 - 3y^2$ soit petite), pour écrire un programme il faut choisir un paramètre N et chercher les solutions (x, y) avec x et y inférieurs à N. De plus, il apparait que si (x, y) est une solution alors on obtient de nouvelles solutions en remplaçant x par $-x$, ou aussi y par $-y$. Bref, on cherche des solutions avec $0 \leqslant x \leqslant N$ et $0 \leqslant y \leqslant N$.

1. Écrire une fonction `pell_fermat(N)` qui *affiche* les solutions trouvées (x, y) à l'équation de Pell-Fermat avec $0 \leqslant x \leqslant N$ et $0 \leqslant y \leqslant N$.
2. Améliorer la fonction pour qu'elle renvoie la liste des couples (type `tuple`) de solutions trouvés.

Rappelons que les syntaxes pour accéder au i-ème élément d'une liste, et au i-ème caractère d'une chaine, sont les mêmes : on peut utiliser une boucle `for` pour parcourir uns par uns tous les caractères d'une chaine.

Exercice 9
Pour un caractère seul `x`, la méthode `x.isupper()` renvoie un booléen `True` si `x` est en majuscule, et `False` sinon.

Écrire un fonction `acronyme(s)` qui prend en argument une chaine de caractères `s` et qui renvoie la liste de tous les caractères majuscules de `s`.

Exemple :

```
>>> acronyme("Biologie, Chimie, Physique et Sciences de la Terre")
['B', 'C', 'P', 'S', 'T']
```

Bonus : pour une *liste* `L` de *chaines de caractères*, la méthode `"".join(L)` colle tous les termes de la liste en une seule chaine de caractères (la syntaxe signifie ici « joindre les éléments de `L` autour de la chaine vide `""` », qu'on pourrait remplacer par n'importe quel caractères de séparation comme `" "` ou `", "`). Reprendre la fonction en renvoyant non pas une liste mais une chaine de caractères.

L'exercice suivant résume bien tout, peut-être à traiter à la toute fin du TP.

Exercice 10 (*)
Le **crible d'Eratosthène** est une ancienne méthode pour trouver tous les nombres premiers jusqu'à n. Il fonctionne de la façon suivante (on rappelle que 0 et 1 ne sont pas des nombres premiers) :

- On écrit tous les nombres à la suite, de 2 à n,
- On barre tous les multiples de 2, sauf 2,
- On barre tous les multiples de 3, sauf 3,
- On avance à 5 (car 4 est barré), puis on barre tous les multiples de 5,
- Et ainsi de suite, on avance au premier nombre non barré (qui est donc premier) et on barre tous ses multiples.

On souhaite s'inspirer de cette méthode pour générer la liste de tous les nombres premiers inférieurs ou égaux à n.

1. Dans un premier temps on modélise le problème avec une liste `L` de booléens `True` ou `False`, où on interprète `L[i] = False` comme « l'entier i est barré » et donc `L[i] = True` comme « non-barré ». Pour bien aligner les indices, on décide que `L[0]` correspond bien au nombre 0 (même si on sait qu'il n'est pas premier ; on peut donc dès le début le mettre à `False`) et de même `L[1]` correspond à 1 qui est aussi `False`. La liste `L` est donc de longueur $n+1$.

   Écrire une fonction `crible(n)` qui renvoie ainsi la liste de booléens correspondant aux nombres premiers inférieurs à n.

   Comme les premiers nombres premiers sont 2, 3, 5, 7, 11, la liste résultat doit commencer par

   ```
   [False, False, True, True, False, True, False, True, False, False, ...]
   ```

2. En déduire une fonction `liste_premiers(n)` qui, à l'aide de la précédente, renvoie la liste de tous les nombres premiers inférieurs ou égaux à `n`.

V Méthode : listes en compréhension

Une autre méthode pour créer des listes est d'utiliser la syntaxe **en compréhension**, qui est plus proche du langage mathématique. Par exemple ceci crée la liste des carrés des nombres de 1 à 10 :

```
>>> [i**2 for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Cela ressemble vraiment beaucoup beaucoup à $\{i^2 \mid 1 \leqslant i < 11\}$ n'est-ce pas ? On peut même y rajouter une condition, par exemple pour avoir seulement les carrés des nombres pairs :

```
>>> [i**2 for i in range(1, 11) if i%2 == 0]
[4, 16, 36, 64, 100]
```

Cette syntaxe a en fait de nombreux avantages. Au moins, c'est la méthode la plus simple pour créer une liste dont on aurait une « formule » directe pour `L[i]` (ce qui n'est pas le cas dans la partie III). Une autre façon standars de créer une liste de n zéros est

```
[0 for _ in range(n)]
```

Exercice 11
Produire la liste suivante, en utilisant une syntaxe en compréhension :

```
[1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0]
```

Exercice 12
Écrire une fonction `rebours(n)` qui, en utilisant directement une liste en compréhension, renvoie la liste constituée de `[n, n-1, ..., 1, 0]` (compte à rebours à partir de n).

VI D'autres opérations sur les listes

VI.1 Tranches

Étant donnée une liste `L`, les syntaxes suivantes permettent d'obtenir une liste extraite de `L` appelée **tranche** :

- `L[a:b]` : sélectionne tous les éléments de la liste d'indice entre `a` et `b` (`b` est **exclus**, comme dans `range(a, b)`).
- `L[a:]` : sélectionne tous les éléments à partir de l'indice `a`.
- `L[:b]` : sélectionne tous les éléments du début jusqu'à l'indice `b` exclus.

Testons par exemple :

```
>>> L = ['oeufs', 'pain', 'riz', 'beurre', 'fromage', 'pommes']
>>> L[1:4]
['pain', 'riz', 'beurre']
>>> L[2:]
['riz', 'beurre', 'fromage', 'pommes']
```

Ces syntaxes sont compatibles avec les indices négatifs. Par exemple `L[:-1]` correspond à toute la liste sauf le dernier élément, et `L[-k:]` correspond aux `k` derniers éléments de la liste.

Enfin, un dernier paramètre optionnel `L[a:b:r]` permet de trancher en « sautant de `r` » au lieu de 1, pour par exemple prendre un terme sur deux. Avec un argument négatif, on part de la fin. Ainsi `L[::-1]` correspond exactement à la liste `L` rangée en ordre inverse (départ de la fin, jusqu'au début, en sautant de -1).

Ces opérations fonctionnent exactement de la même manière sur les chaines de caractères, si on les considère comme des listes de caractères individuels, illustrons-les encore :

```
>>> s = "mathématiques"
>>> len(s)
13
>>> s[0]
'm'
>>> s[-1]
's'
>>> s[:4]
'math'
>>> s[1:-1]
'athématique'
>>> s[::-1]
'seuqitaméhtam'
```

Exercice 13
Pouvez-vous prédire ce que renvoie la ligne suivante ?

```
>>> s = "J'aime la biologie, la chimie et les mathématiques"
>>> s[:7] + s[-17:]
```

VI.2 Itération sur les éléments

Pour parcourir une liste, il existe une autre syntaxe qui s'appelle **itérer sur les éléments**. La syntaxe `for x in L` fournit uns par uns les éléments de `L` dans la variable `x` :

```
L = ["oeufs", "pain", "riz", "beurre"]
for x in L:
    print(x)
```

produit tout simplement

```
oeufs
pain
riz
beurre
```

C'est souvent *plus simple* et *plus élégant*, même si en pratique itérer sur les indices fonctionne toujours.

L'itération sur les éléments fonctionne aussi sur les chaines de caractères, en fournissant les caractères uns par uns :

```
s = "BCPST"
for x in s:
    print(x)
```

produit la même chose que l'itération sur les indices

```
B
C
P
S
T
```

La boucle `for` en Python fait beaucoup plus de choses que répéter n fois et c'est une possibilité fort intéressante et subtile. On parle d'**objets itérables** pour désigner tous les objets qu'on peut mettre après un `for` et qui sont capables de fournir des éléments uns par uns. Dans le TP précédent nous avons brièvement parlé d'itération sur les éléments d'un tuple.

On peut même combiner cela dans les listes en compréhension. La syntaxe suivante prend une liste `L` et fabrique une liste `M` dont les éléments sont exactement les doubles de ceux de `L` :

```
M = [2*x for x in L]
```

La fonction `garde_positifs(L)` de la partie IV peut s'écrire plus simplement avec une seule ligne

```
P = [x for x in L if x >= 0]
```

Exercice 14 (*)
Reprendre les autres exercices de la partie IV en utilisant uniquement des listes en compréhension.