TP 5
Listes

I Rappel des épisodes précédents

Les types vus jusque là :
- `int` : nombres entiers
- `float` : nombres à virgule flottante (virgule `.` et notation scientifique avec `e`)
- `str` : chaines de caractères, entre guillemets doubles `"`
- `bool` : booléens, avec `True` et `False`
- Connaitre le type : fonction `type()` (mode interactif)
- Conversions de type : fonctions `int()`, `float()`, `str()`, `bool()`
- Remarque : les fonctions sont aussi des objets du type... `function`
- `tuple` : produit cartésien, et la conversion de type `tuple()`
- `range` : type intervalle, créé par la fonction `range()`

Les opérations :
- Les 4 opérations `+`, `-`, `*`, `/`
- Le quotient dans la division euclidienne (en nombres entiers) `//`, le reste `%`
- Les comparaisons `==`, `!=`, `<`, `<=`, `>`, `>=`
- Les opérations sur les booléens `and`, `or`, `not`
- Les opérations sur les chaines de caractères :
  - concaténation `+`
  - multiplication par un nombre entier `*`
  - accès au `i`-ème caractère de `s` (démarrant à `0`) par `s[i]`
  - longueur de la chaine `len(s)`

Les fonctions :
- Afficher : `print()`
- Demander à l'utilisateur : `input()`, mais **nous ne l'utilisons plus** car nous écrivons des fonctions avec des paramètres.
- Aide sur une fonction ou un module : `help()` (mode interactif)
- Assertion : `assert()`, fait échouer le programme si la condition n'est **pas** vérifiée.

Les conditions (les `elif` et le `else` ne sont pas du tout obligatoires) :

```
if condition1:
    instructions1
elif condition2:
    instructions2
elif condition3:
    instructions3
...
else:
    instructions
```

La boucle `while` :

```
while condition:
    bloc d'instructions
```

Les déclarations de fonctions :

```
def nom(argument1, argument2, ...):
    """ documentation """
    bloc d'instructions
```

avec l'instruction `return` qui arrête l'exécution et éventuellement renvoie une valeur.

À partir de maintenant il faut être indépendant quant à la manipulation du mode interactif ou du mode script, savoir passer de l'un à l'autre pour écrire des fonctions ou pour les tester.

II Cours

II.1 Les bases

Les listes en Python contiennent un certain nombre d'éléments, rangés les uns à la suite des autres, écrits entre crochets et séparés par des virgules. Ce sont des types comme les autres, pouvant rentrer dans des variables ou dans l'instruction `print`. 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 pas forcément tous du même type ! Voici par exemple une liste de courses :

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

À tout moment on peut accéder au `i`-ème élément de la liste, et aussi le modifier :

```
L[1] = 12
print(L)
```

Les éléments sont numérotés à partir de 0 !

II.2 Accès aux éléments

Pour une liste `L`, le `i`-ème élément est obtenu par `L[i]`. La fonction `len()` donne la longueur de la liste. Si la liste est de longueur `n` alors les éléments sont numérotés de `0` à `n-1` (cela fait bien `n` éléments !). De plus les indices négatifs permettent de revenir en arrière : `L[-1]` est le dernier élément de la liste, `L[-2]` l'avant dernier etc.
Les syntaxes suivantes permettent d'obtenir une liste appelée *tranche* de `L` :

- `L[a:b]` : sélectionne tous les éléments de la liste d'indice entre `a` et `b` (`b` est **exclus** : même problème que 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.

Testez sur vos propres exemples de listes !

Exercice 1. Ces syntaxes se combinent aussi avec les indices négatifs. Alors, comment sélectionner les `k` derniers éléments de la liste ? Et sélectionner tous les éléments sauf les `k` derniers ?

II.3 Parcourt d'une liste

Pour parcourir une liste (= effectuer une opération sur chacun de ses éléments) il y a deux possibilités.
La première s'appelle **itérer sur les indices**, par exemple

```
for i in range(len(L)):
    print(L[i])
```

Ce n'est qu'une boucle classique où `i` varie entre `0` et `len(L)-1` (ce qui est bien le dernier élémént de la liste).
Mais dans de nombreuses situations on a besoin seulement de récupérer les éléments de la liste uns pas uns mais sans connaitre la variable `i` ni d'effectuer d'opération qui va modifier `L[i]` : on dit alors qu'on **itère sur les éléments** avec la syntaxe

```
for x in L:
    print(x)
```

C'est dans de nombreux cas *plus simple* et *plus élégant* et il vous faudra bientôt choisir laquelle des deux possibilités est la plus adaptée...
Dans tous les cas, à n'importe quel moment `print(L)` affiche d'un seul coup la liste `L`. Cela permet de rapidement inspecter le contenu d'une liste au fur et à mesure du programme.

Remarque 1. La syntaxe `for x in ...` fonctionne pour les listes, les chaines de caractères, les tuples, les intervalles (`range`)... Tous ces objets sont appelés en Python des **objets itérables**, sur lesquelles une boucle `for` permet d'obtenir des éléments uns par uns. Le mécanisme de la boucle `for` et des objets itérables est plus subtil qu'il n'y parait et c'est une fonctionnalité de Python particulièrement intéressante, qu'il faut apprendre à maitriser. On peut même fabriquer ses propres objets itérables (niveau : avancé) !

II.4 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`.
Pour un nombre entier `n` il y a aussi une opération du multiplication `*` entre une liste et `n` : le résultat `L * n` est la même chose que `L + L + ... + L` (`n` fois).
Testez sur vos propres exemples !

```
["oeuf", "pain"] + ["riz", "pates"]
["pile", "face"] * 10
```

Exercice 2.
1. Quelle est la longueur de `L + M` ? Et de `L * n` ?
2. Si un élément `x` a pour indice `i` dans `M` alors à quel indice le retrouve-t-on dans `L + M` ?
3. Si un élément `x` a pour indice `i` dans `L` alors à quel(s) indice(s) le retrouve-t-on dans `L * n` ?

Pour ajouter un élément seul `x` à la liste `L`, on utilise la syntaxe `L.append(x)`. Ceci ne retourne rien, mais modifie la liste elle-même.

Remarque 2. 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 a accès à la liste `L` et en modifie la structure en profondeur (pour une autre liste `M` ce sera `M.append(x)`). On parle de *méthode* plutôt que de fonction, et c'est un concept de base de la *programmation orientée objet* : chaque objet (variable entière, chaine de caractère, liste, ...) vient avec des méthodes qui modifient l'objet lui-même. Le langage Python utilise abondamment ces concepts (ah bon ? essayez `help(list)` et lisez), même si nous pouvons aussi largement nous en passer pour l'instant.
On pourrait aussi penser faire `L = L + [x]` qui donnera bien le même résultat mais 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,

et c'est beaucoup plus lourd à comprendre pour l'ordinateur que de dire : modifier `L` pour lui ajouter un élément `x`.

À l'inverse, la méthode `L.pop()` supprime le dernier élément de la liste **et le retourne**, donc on peut écrire `x = L.pop()` pour le récupérer avant qu'il ne disparaisse.

Remarque 3. C'est encore une *méthode* de la liste `L`. L'idée est inspirée par la notion en de *pile* en informatique : c'est une façon de stocker des objets dans laquelle on peut seulement les empiler les uns sur les autres, donc à tout moment ajouter un élément sur le haut de la pile (opération appelée `push`) ou récupérer l'élément sur le haut (opération `pop`), sans se soucier de l'indice ou du nombre d'éléments.

Testez sur vos propres exemples !

```
L = [1, 3, 5, 7]
L.append(9)
print(L)
L.pop()
L.pop()
# continuez jusqu'à vider la liste !
```

II.5 Obtenir des listes

Pour créer de nouvelles listes on a les possibilités suivantes :

1. Lister les éléments uns par uns :

   ```
   L = [x, y, ...]
   ```

2. Créer une nouvelle liste de taille `n` remplie avec tous les mêmes éléments, par exemple le nombre `0` :

   ```
   L = [0] * n
   ```

3. Utiliser la conversion de type `list()` depuis un type `range` :

   ```
   L = list(range(1, 11))
   ```

   ou aussi depuis le type `str` :

   ```
   list("Bonjour")
   ```

4. Partir d'une liste vide et itérer la méthode `append` pour ajouter les éléments uns par uns. Contrairement aux cas précédents qui étaient pratiques pour obtenir directement une liste dont la longueur est connue à l'avance (quitte à en modifier plus tard le contenu) on l'utilise plutôt quand on ne sait pas tout de suite quelle taille la liste aura, par exemple on étudie une équation et on rajoute au fur et à mesure les solutions qu'on trouve. Par exemple ceci fournit la liste des entiers $i \geqslant 0$ tels que $i^3 \leqslant 100$:

```
L = []
i = 0
while i**3 <= 100:
    L.append(i)
    i = i + 1
```

5. Définir les listes **en compréhension**, par exemple ceci génère la liste des carrés des nombres de 1 à 10 :

```
L = [i**2 for i in range(1, 11)]
```

et on peut même rajouter une condition, par exemple pour avoir seulement les carrés des nombres pairs :

```
L = [i**2 for i in range(1, 11) if i%2 == 0]
```

Cela ressemble vraiment beaucoup beaucoup à $\{i^2 \mid i \in \mathbb{Z} \text{ et } 1 \leqslant i < 11\}$ n'est-ce pas ?

III Exercices

Exercice 3. Écrire des fonctions prenant en argument un nombre n et qui génèrent les listes suivantes (écrites en notation mathématiques) en utilisant uniquement la notation en compréhension (5 ci-dessus). Une seule ligne doit suffire ! De la forme

```
def A(n):
    return [... for ... in ...]
```

1. `A(n)` : $\big(n, n-1, \dots, 1, 0\big)$
2. `B(n)` : $\big(0, \frac{1}{n}, \frac{2}{n}, \dots, \frac{n-1}{n}, 1\big)$
3. Importer les fonctions `sin()`, `cos()`, et la constante `pi` de la librairie `math` avec

   ```
   from math import sin, cos, pi
   ```

   puis `C(n)` génère la liste des couples $\big(\cos(\frac{k\pi}{2n}), \sin(\frac{k\pi}{2n})\big)$ pour $0 \leqslant k \leqslant n$.
4. Importer la fonction `randint` du module `random` et lire sa documentation. Puis `D(n)` génère une liste de `n` nombres aléatoires correspondant à une suite de `n` lancers de dés à six faces.

On s'intéresse maintenant aux listes correspondant aux termes successifs d'une suite définie par récurrence. L'intérêt des listes est qu'on garde en mémoire d'un seul coup tous les termes précédents : ainsi une boucle telle que

```
u = # terme initial u_0
i = 0
for i in range(n):
    u = f(u) # u devient u_(i+1) donné par f(u_i)
```

devient

```
L = [0] * n
L[0] = # terme initial
for i in range(1, n): # attention, on va écrire L[i-1] donc i >= 1
    L[i] = f(L[i-1])
```

et dans les récurrences doubles cela est nettement plus simple !

Exercice 4. Écrire des fonctions prenant en argument un nombre n et qui retournent les listes composées des n premiers termes (c'est à dire les termes d'indice de 0 à $n-1$) des suites suivantes. Dès la première ligne la fonction initialise une liste `L` de n éléments (méthodes 2 ou 3 ci-dessus), puis une boucle modifie les éléments.

1. `E(n)` : factoriel *(encore ? ? ?)*, la liste initiale est remplie de zéros, et à la fin `L[i]` est le produit $1 \times 2 \times \cdots \times i$, et pour $i = 0$ c'est 1.
2. `F(n)` : Fibonacci *(encore encore ? ? ?)*, la liste initiale est remplie de zéros, et à la fin `L[i]` est le `i`-ème terme de la suite de Fibonacci $(F_i)_{i\in\mathbb{N}}$ avec $F_0 = 0$, $F_1 = 1$ et $\forall i \geqslant 0$, $F_{i+2} = F_{i+1} + F_i$.

Dans les exercices suivants on génère les listes en partant d'abord d'une liste vide puis en itérant la méthode `append` comme en 4 car on ne sait pas à l'avance combien d'éléments la liste aura.

Exercice 5. Écrire une fonction `positifs(L)` qui prend en argument une liste de nombres `L` et retourne une liste `P` composée uniquement des nombres de `L` qui sont positifs.

Exercice 6. Écrire un fonction `acronyme(s)` de la façon suivante : la fonction a pour argument une chaine de caractères, de type `str`, lit les caractères uns par uns, et génère la liste `M` de toutes les majuscules présentes dans `s` (on oublie les autres lettres). Pour un caractère `x`, la méthode `x.isupper()` retourne un booléen et permet de savoir si `x` est en majuscule ou non. Puis tester :

```
acronyme("Biologie, Chimie, Physique et Sciences de la Terre")
```

Bonus : retourner non pas `M` mais `"".join(M)`

Exercice 7. *Équation de Pell-Fermat*
On recherche des solutions de l'équation suivante :

$$n^2 - 3m^2 = 1, \quad (n, m) \in \mathbb{Z}^2 \tag{1}$$

Comme il pourrait y avoir une infinité de solutions (au contraire de $n^2 + 3m^2 = \dots$, où le terme devient trop grand dès que n ou m est trop grand, il se peut que n et m soient tous les deux extrêmement grands et que pourtant la différence $n^2 - 3m^2$ soit petite), pour écrire un programme il faut choisir un paramètre N et chercher les solutions (n, m) avec n et m inférieurs à N. De plus, il apparait que si (n, m) est une solution alors on obtient de nouvelles solutions en remplaçant n par $-n$, ou aussi m par $-m$. Bref, on cherche des solutions avec $0 \leqslant n \leqslant N$ et $0 \leqslant m \leqslant N$.
Écrire une fonction `pell_fermat(N)` qui prend en argument une telle borne N et retourne la liste des couples (n, m) trouvés comme solutions de l'équation de Pell-Fermat et la tester avec des nombres N de plus en plus grands.

Remarque 4. Dans les exercices précédents on peut en fait s'en sortir en une seule ligne en compréhension de la forme

```
return [... for ... in ... if ...]
```

voire même avec un double `for`

```
return [... for n in ... for m in ... if ...]
```

C'est bien sur une possibilité extrêmement puissante... pouvez-vous réécrire les fonctions ainsi ? Si non, on s'en passera bien aussi : cela semble trop éloigné de l'idée naturelle de rajouter les éléments uns par uns.

Un exercice pour s'occuper qui résume tout.

Exercice 8. 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.

Écrire une fonction `crible(n)` inspirée de cette idée qui retourne une liste `L` composée de `True` ou de `False`, où on interprète `L[i] = False` comme « on a barré le nombre `i` ». On décide que `L[0]` correspond bien au nombre 0 (même si nous n'avons pas du tout besoin de 0 ni de 1, c'est beaucoup plus pratique ainsi) et donc dès le départ `L[0] = False` et `L[1] = False` et `L` contient $n + 1$ élements. Le programme peut être très court, mais attention aux indices des boucles !
Dans un deuxième temps, écrire une fonction `print_crible(n)` qui récupère le résultat de `crible(n)` puis affiche seulement les nombres non barrés.
Dans un troisième temps, écrire une fonction `liste_premiers(n)` qui récupère le résultat de `crible(n)` puis génère précisément la liste des nombres premiers inférieurs à n.