Le but de ce TP est de se remettre à jour sur toutes les bases de Python.

16.1 Les listes

16.1.1 Définition par compréhension

Imaginons qu'on veut créer une liste `L` à partir d'un *itérable* (par exemple `range(10, 20)`, ou bien une autre liste), avec éventuellement des conditions de filtrage. La structure générale pour cela est

```
L = []
for x in iterable:
    if condition_sur_x:
        L.append(fonction_de_x)
```

C'est tout de suite plus clair avec un exemple. Créons la liste `[1, 9, 25, 49, 81]`, c'est-à-dire des n^2 lorsque n est impair, pour n entre 1 et 9. Cela s'écrit

```
L = []
for n in range(1, 10):
    if n % 2 == 1:
        L.append(n ** 2)
```

La définition par compréhension permet d'écrire ce bloc de code en une seule ligne

```
L = [fonction_de_x for x in iterable if condition_sur_x]
```

Ou pour notre exemple

```
L = [n**2 for n in range(1, 10) if n % 2 == 1]
```

Ce n'est pas difficile de se souvenir de la syntaxe, cela suit la language naturel : "donne moi les n^2 pour $n \in \{1, \dots, 9\}$, mais seulement si n est impair".

1. a) Créer en une ligne la liste `L` des entiers divisibles par 2 ou 3 de l'intervalle $\{0, 100\}$.

b) Créer avec le plus petit code possible la liste `T`

```
[
    [0, 10, 20, ..., 100],
    [1, 11, 21, ..., 91],
    [2, 12, 22, ..., 92],
    ...,
    [9, 19, 29, ..., 99]
]
```

16.1.2 Les slices

Il existe en Python une syntaxe pour obtenir les éléments à certains indices d'une liste. Ce sont les *slices*. On rappelle la signification de `range(a,b,c)` et ses variantes

- `range(a,b,c)` produit les entiers $a, a+c, a+2c, \dots$ jusqu'à b exclus. Par exemple, `range(11, 23, 4)` produit les entiers $11, 15, 19$, c'est-à-dire qu'on part de 11, on avance de 4 en 4, et si on atteint ou dépasse 23 on s'arrête avant.
- `range(a,b)` produit les entiers $a, a+1, \dots, b-1$. Autrement dit, si on omet c, Python considère que $c = 1$.
- `range(b)` produit les entiers $0, 1, \dots, b-1$. Autrement dit, si on omet a, Python considère que $a = 0$.

À noter que c peut être négatif.

De même, si `L` est une liste (ou plus généralement un *itérable*), on peut écrire `L[a:b:c]` pour accéder à certains éléments de `L`. Plus précisément

- `L[a:b]` produit la liste des éléments d'indice $a, a+1, \dots, b-1$
- `L[:b]` produit la liste des éléments d'indice $0, 1, \dots, b-1$. Autrement dit, si on omet a, Python part du début de la liste.
- `L[a:]` produit la liste des éléments d'indice $a, a+1, \dots, len(L)-1$. Autrement dit, si on omet b, Python va jusqu'au bout de la liste.

— `L[a:b:c]` produit la liste des éléments d'indice $a, a+c, a+2c, \ldots$ jusqu'à b exclus. Par exemple, `L[11:23:4]` produit la liste des éléments de `L` d'indice $11, 15, 19$, c'est-à-dire qu'on part de l'indice 11, on avance de 4 en 4, et si on atteint ou dépasse 23 on s'arrête avant.

À noter que là encore, c peut être négatif. Si c est négatif, la valeur par défaut de a et b est inversée (c'est-à-dire que si on omet a, Python considère que c'est la fin de la liste, et si on omet b Python considère que c'est le début de la liste).

2. a) Que fait `L[::]` ?

b) Que fait `L[::-1]` ?

c) Que vaut `[1, 3, 5, 7][1:2]` ?

d) À partir de `T` défini à l'exercice précédent, créer la liste

```
[
    [1, 11, 21, ..., 91],
    [3, 13, 23, ..., 93],
    ...,
    [9, 19, 29, ..., 99]
]
```

Les syntaxes les plus importantes à retenir sont celles sans c, c'est-à-dire `L[:b]`, `L[a:]` et `L[a:b]`.

Le slicing n'est pas spécifique aux listes, on peut utiliser la même syntaxe pour, par exemple, les chaînes de caractères.

3. a) Écrire en deux lignes une fonction `prefixe(p, s)` qui renvoie `True` si la chaîne de caratères `p` est un préfixe de la chaîne de caractères `s`, (c'est-à-dire que `s` "commence par" `p`), `False` sinon.

b) Écrire en deux lignes une fonction `suffixe(p, s)` qui renvoie `True` si la chaîne de caratères `p` est un suffixe de la chaîne de caractères `s`, (c'est-à-dire que `s` "finit par" `p`), `False` sinon.

c) Le *bord* d'une chaîne de caractères `s` est la plus grande chaîne de caractères à la fois préfixe et suffixe de `s`, mais qui n'est pas `s` elle-même. Éventuellement, le bord peut être la chaîne de caractères vide `""`. Écrire une fonction `bord(s)` qui trouve le bord de `s`.

16.2 Les booléens

Les valeurs booléennes en Python sont `True` et `False`. On peut les composer avec les opérateurs `and`, `or` et `not`. De même que la multiplication a priorité sur l'addition, `and` a priorité sur `or`.

4. a) Que vaut `True and False` ?

b) Que vaut `True and False or True and False or True and False` ?

c) Que vaut `True and (False or True) and (False or True)` ?

d) Écrire en deux lignes une fonction `xor(b1, b2)` qui est le "ou exclusif", c'est-à-dire qui renvoie `True` si `b1` est vrai ou `b2` est vrai, mais pas les deux à la fois.

e) Écrire en deux lignes une fonction `nand(b1, b2)` qui est le "tout sauf tout le monde", qui renvoie `False` si à la foin `b1` et `b2` sont vrais, et `True` sinon.