TP #3 — Jouer avec les mots

L'objectif de ce TP est de manipuler les chaînes de caractères en Python et —chose exceptionnelle— utiliser la fonction `print` (il faudra cependant bien avancer pour cela) !

On rappelle qu'une chaîne de caractères « `str` » est similaire à un `tuple` dont les éléments sont tous eux-mêmes des chaînes de caractères de longueur 1. Notamment, on peut parcourir tous ces « caractères » d'une chaîne `s` avec l'énumération `for c in s`. Il existe de nombreuses fonctions définies uniquement pour les chaînes de caractères ; nous en utiliserons une dans le dernier exercice.

Toutes les fonctions qu'on demande d'écrire doivent bien évidemment être testées.

Exercice 1. *Sur l'alphabet* $\{a, b\}$

1. Écrivez une fonction `isOnlyAB(w)` qui prend en entrée une chaîne de caractère `w` et qui renvoie `True` si `w` est un *mot* sur l'*alphabet* $\{a, b\}$, c'est à dire si `w` est la chaîne vide (ou nulle) ou si ses seuls caractères sont `'a'` ou `'b'`, et `False` sinon.
2. Écrivez une fonction `evenA(w)` qui renvoie `True` si `w` est un mot sur $\{a, b\}$ comportant un nombre pair de a, et `False` sinon. En dehors d'une éventuelle variable de boucle, cette fonction ne devra utiliser qu'une unique variable, qui devra être de type `bool`
3. Écrivez une fonction `AnBn(w)` qui renvoie `True` si `w` est de la forme « $a^n b^n$ » pour $n \in \mathbb{N}$ quelconque, c'est à dire est un mot formé de n caractères a suivis de n caractères b, et `False` sinon.

Exercice 2. *Facteurs et sous-mots*

On définit la concaténation de deux mots $u = u_0 \cdots u_n$ (dont les lettres sont u_0, …, u_n) et $v = v_0 \cdots v_m$ comme le mot $u \cdot v := u_0 \cdots u_n v_0 \cdots v_m$

1. Écrivez une fonction `isPrefix(p, w)` qui prend en entrée deux chaînes de caractère `p` et `w` représentant chacune un mot sur un certain alphabet, et renvoie `True` si `p` est un *préfixe* de `w`, et `False` sinon. Un mot p (éventuellement nul) est préfixe d'un mot w ssi. il existe un mot s tel que $w = p \cdot s$. Autrement dit pour $p = p_0 \cdots p_n$, $w = w_0 \cdots w_{m \geq n}$, p est préfixe de w ssi. $\forall\, i \in [\![0, n]\!]$, $p_i = w_i$.
2. Écrivez une fonction `isFact(f, w)` qui prend en entrée deux chaînes de caractères `f` et `w` représentant chacune un mot sur un certain alphabet, et renvoie `True` si `f` est un *facteur* de `w`, et `False` sinon. Un mot f (éventuellement nul) est facteur d'un mot w ssi. il existe deux mots (éventuellement nuls) p et s tels que $w = p \cdot f \cdot s$.

Vous pourrez vous contenter ici d'une approche naïve, pas nécessairement efficace en fonction des longueurs de w et f.
Conseil : on rappelle que pour toute chaîne `s`, la *tranche* de `s` formée des « caractères » d'indice i (inclus) à j (exclu) peut s'obtenir en Python avec l'expression `s[i:j]`.

3. Écrivez une fonction `isSW(sw, w)` qui prend en entrée deux chaînes de caractères `sw` et `w` représentant chacune un mot sur un certain alphabet, et qui renvoie `True` si `sw` est un *sous-mot* de `w`, et `False` sinon. Un mot u' est sous-mot de u s'il est une sous-suite de u : en écrivant $|w|$ le nombre de lettres d'un mot et en indiçant celles-ci à partir de zéro, on a $|u'| = 0$ ou $\forall i \in [\![0, |u'|-1]\!]$. $u'_i = u_{\varphi(i)}$ pour une certaine fonction $\varphi : [\![0, |u'|-1]\!] \to [\![0, |u|-1]\!]$ strictement croissante.

Exercice 3. *Palindromes et mots « de Fibonacci »* ⋆

On définit le *miroir* d'un mot non nul $w = w_0 \cdots w_n$ comme $\widehat{w} := w_n \cdots w_0$, et le miroir du mot nul comme lui-même. Un *palindrome* est un mot w égal à son miroir $\widehat{w}$.

1. Écrivez une fonction itérative `isPalinI(w)` qui prend en entrée une chaîne de caractères `w` représentant un mot sur un certain alphabet et renvoie `True` si `w` est un palindrome, et `False` sinon.
2. Écrivez une fonction récursive `isPalinR(w)` de même spécifications que `isPalinI`.

On définit les mots « de Fibonacci » sur l'alphabet $\{a, b\}$ par la récurrence suivante :

— $f_0 = a$
— $f_1 = b$
— $f_{n+2} = f_{n+1} \cdot f_n$

3. Écrivez une fonction `fw(n)` qui calcule et renvoie une chaîne de caractères représentant le n-ième mot de Fibonacci. Votre fonction doit être suffisamment efficace pour pouvoir calculer f_{40} quasiment instantanément.
4. Montrez que pour tout n pair (resp. impair) ≥ 2, f_n se termine par ba (resp. ab).
5. Montrez que le mot g_n obtenu à partir de f_n en en supprimant les 2 dernières lettres est un palindrome.
6. Écrivez une fonction `testGPalin(n)` qui vérifie la propriété ci-dessus pour les n premiers mots g_n.

Exercice 4. *Mots bien parenthésés* ⋆

On considère un alphabet comportant des paires de caractères de parenthèses comprenant une parenthèse ouvrante (par ex. « (») et une parenthèse fermante (par ex. «) »).
Informellement, Un *mot bien parenthésé* sur un tel alphabet est un mot tel que :

— à toute parenthèse ouvrante correspond une parenthèse fermante dans le mot (le nombre total de parenthèses ouvrantes et fermantes dans le mot est le même) ;
— on ne peut fermer une parenthèse que si le nombre de parenthèses ouvertes (en partant de la première lettre) qui n'ont pas déjà été fermées est non nul.

Formellement, on peut donner la définition inductive suivante :

— cas de base :
  — le mot vide est bien parenthésé
  — a est un mot bien parenthésé pour toute lettre a qui n'est pas une parenthèse
  — () est un mot bien parenthésé (ce cas n'est pas strictement nécessaire)
— cas inductifs (on pourrait fusionner les deux cas en un seul) :
  — si w est un mot bien parenthésé, alors (w) est bien parenthésé
  — si w_1 et w_2 sont deux mots bien parenthésés, alors $w_1 \cdot w_2$ est bien parenthésé

On donne dans les Fig. 1 ~ 3 un exemple de mot bien parenthésé et deux exemples de mots mal parenthésés (pour des raisons différentes) :

FIGURE 1 – Un mot bien parenthésé

FIGURE 2 – Un mot mal parenthésé

FIGURE 3 – Un autre mot mal parenthésé

1. Écrivez une fonction `wp1(w)` qui prend en entrée une chaîne de caractère `w` représentant un mot sur l'alphabet { `'('`, `')'` } (ou si vous le souhaitez, un alphabet quelconque incluant ces deux lettres) et renvoie `True` si `w` est bien parenthésé, et `False` sinon.

On peut étendre la notion de mot bien parenthésé à un alphabet comportant plusieurs types de parenthèses (par exemple en ajoutant « [» et «] »): il suffit pour cela de généraliser la définition inductive en ajoutant un cas inductif pour chaque type supplémentaire de paires de parenthèses. Il devient cependant plus complexe d'efficacement déterminer si un mot est bien parenthésé, car on ne peut plus se contenter de compter le nombre de parenthèses ouvertes pour chaque type de parenthèse. En effet le mot ([)] ferme bien chacune de ses parenthèses et ne ferme pas une parenthèse avant qu'elle ne soit ouverte, mais n'est cependant pas bien parenthésé. On donne un autre exemple de mot mal parenthésé avec plusieurs types de parenthèses à la Fig. 4.

FIGURE 4 – Encore un mot mal parenthésé

Un algorithme efficace pour déterminer si un mot avec plusieurs types de parenthèses est bien parenthésé consiste à utiliser une structure de données de *pile*. Celle-ci peut s'implémenter en Python avec les valeurs et opérations suivantes :

- Une pile vide peut se définir comme une liste vide `[]`
- *Empiler* un élément `x` sur une pile `p` consiste à ajouter cet élément en fin de liste, grâce à `append`
- *Dépiler* un élément consiste à retirer l'élément en fin de liste et le renvoyer, grâce à `pop`.

Pour déterminer si un mot est bien parenthésé, il suffit alors de créer une pile vide puis de parcourir tous ses caractères du début à la fin et :

- si l'on rencontre une parenthèse ouvrante, on l'empile sur la pile
- si l'on rencontre une parenthèse fermante, on dépile l'élément au sommet de la pile et :
  - s'il n'y avait pas d'élément : on renvoie faux (le mot n'était pas bien parenthésé : on a cherché à fermer plus de parenthèses qu'actuellement ouvertes)
  - si la parenthèse (ouvrante) dépilée ne correspond pas à la parenthèse fermante (par exemple on avait une parenthèse fermante «] » et l'on a dépilé une parenthèse ouvrante « { ») : on renvoie faux
  - sinon on continue
- si toutes les lettres ont été parcourues sans causer d'erreur, on vérifie que la pile est vide (c'est à dire qu'on a fermé toutes les parenthèses), et l'on renvoie vrai dans ce cas, faux sinon

2. Écrivez une fonction `wp4(w)` qui prend en entrée une chaîne de caractère `w` représentant un mot sur l'alphabet sur l'alphabet { `'('`, `')'`, `'{'`, `'}'`, `'<'`,

`'>'`, `'['`, `']'` } (éventuellement enrichi d'autres paires de parenthèses de votre choix) et renvoie `True` si `w` est bien parenthésé, et `False` sinon.
Conseil : il pourra être utile d'utiliser un dictionnaire pour définir (par extension) la fonction qui à une parenthèse fermante d'un certain type renvoie la parenthèse ouvrante du même type.

Pour conclure cet exercice on souhaite colorier un mot bien parenthésé en coloriant une paire de parenthèses correspondantes avec la même couleur, et en surlignant la première parenthèse causant une erreur dans le cas d'un mot qui n'est pas bien parenthésé (par exemple comme dans les Fig. 2 ∼ 4 ci-dessus).
Pour changer la couleur d'un caractère dans une chaîne de caractère, vous pouvez préfixer ce caractère du code couleur correspondant et le suffixer du code de remise à la couleur par défaut. Par exemple, pour colorier le caractère `c` en rouge, vous pouvez construire la chaîne de caractères `"\033[1;91m" + c + "\033[0;0m"`.
Le (joli) résultat ne sera visible que si vous **affichez** la chaîne avec `print(c)`.

> Le mécanisme utilisé ici dépendant du système, il se peut cependant qu'il ne fonctionne pas sur la machine que vous utilisez. Ne passez pas trop de temps sur cette question si ce n'est pas le cas...

On donne les exemples de codes couleurs ci-dessous :

```
# remise au défaut
colr = "\033[0;0m"
# surligné
cole = ["\033[7;91m", "\033[7;92m",
        "\033[7;93m", "\033[7;94m",
        "\033[7;95m", "\033[7;96m",
        "\033[7;97m","\033[7;90m" ]
# gras
col = ["\033[1;91m", "\033[1;92m",
       "\033[1;93m", "\033[1;94m",
       "\033[1;95m", "\033[1;96m",
       "\033[1;97m","\033[1;90m"  ]
```

Et un exemple de résultat dans la Fig. 5.

FIGURE 5 – Enfin un mot à nouveau bien parenthésé

3. Écrivez une variante `wp4colour(w)` de votre fonction qui renvoie une chaîne de caractères coloriant `w` comme décrit ci-dessus. Il pourra être pratique d'utiliser la fonction « méthode » `join` (hors programme) de la façon suivante : si `s` est une `list` de `str`, `"".join(s)` renvoie la chaîne de caractère obtenue en concaténant tous les éléments de `s` à la chaîne vide.