Chercher et remplacer

Chercher et remplacer sont deux tâches très fréquentes. Savoir les utiliser et comprendre comment elles fonctionnent te permettra d'être plus efficace.

Activité 1 (Chercher)**.**

Objectifs : apprendre différentes façons de chercher avec `Python`*.*

1. **L'opérateur « in ».**
   La façon la plus simple de savoir si une sous-chaîne est présente dans une chaîne de caractères est d'utiliser l'opérateur « `in` ». Par exemple, l'expression :
   ```
   "PAS" in "ETRE OU NE PAS ETRE"
   ```
   vaut « vrai » car la sous-chaîne **PAS** est bien présente dans la phrase **ETRE OU NE PAS ETRE**.
   Déduis-en une fonction `chercher_in(chaine,sous_chaine)` qui renvoie « vrai » ou « faux », selon que la sous-chaîne est (ou non) présente dans la chaîne.
2. **La méthode `find()`.**
   La méthode `find()` s'utilise sous la forme `chaine.find(sous_chaine)` et renvoie la position à laquelle la sous-chaîne a été trouvée.
   Teste ceci sur l'exemple précédent. Que renvoie la fonction si la sous-chaîne n'est pas trouvée ?
3. **La méthode `index()`.**
   La méthode `index()` a la même utilité, elle s'utilise sous la forme `chaine.index(sous_chaine)` et renvoie la position à laquelle la sous-chaîne a été trouvée.
   Teste ceci sur l'exemple précédent. Que renvoie la fonction si la sous-chaîne n'est pas trouvée ?
4. **Ta fonction `chercher()`.**
   Écris ta propre fonction `chercher(chaine,sous_chaine)` qui renvoie la position de départ de la sous-chaîne si elle est trouvée (et renvoie `None` si elle ne l'est pas).
   Tu n'as pas le droit d'utiliser les fonctions `Python`, tu as seulement droit de tester si deux caractères sont égaux !

Activité 2 (Remplacer)**.**

Objectifs : remplacer des portions de texte par d'autres.

1. La méthode `replace()` s'utilise sous la forme :
   ```
   chaine.replace(sous_chaine,nouv_sous_chaine)
   ```
   Chaque fois que la séquence `sous_chaine` est trouvée dans `chaine`, elle est remplacée par `nouv_sous_chaine`.

Transforme la phrase **ETRE OU NE PAS ETRE** en **ETRE OU NE PLUS ETRE**, puis en **AVOIR OU NE PLUS AVOIR**.

2. Écris ta propre fonction `remplacer()` que tu appelleras sous la forme suivante :
   `remplacer(chaine,sous_chaine,nouv_sous_chaine)`
   et qui remplace seulement la première occurrence de `sous_chaine` trouvée. Par exemple `remplacer("ABBA","B","XY")` renvoie `"AXYBA"`.
   Indication. Tu peux utiliser ta fonction `chercher()` de l'activité précédente pour trouver la position de départ de la séquence à remplacer.
3. Améliore ta fonction pour construire une fonction `remplacer_tout()` qui remplace cette fois toutes les occurrences rencontrées.

Cours 1 (Expressions rationnelles *regex*)**.**
Les ***expressions rationnelles*** permettent de chercher des sous-chaînes avec une plus grande liberté : par exemple on autorise un caractère joker ou bien plusieurs choix possibles pour un caractère. Il existe plein d'autres possibilités, mais nous étudions seulement ces deux-là.

1. On s'autorise une lettre joker symbolisée par un point « **.** ». Par exemple si on cherche l'expression « **P . R** » alors :
   - **PAR**, **EMPIRE**, **PURE**, **APPORTE** contiennent ce groupe (par exemple pour **PAR** le point joue le rôle de **A**),
   - mais pas les mots **CAR**, **PEUR**, **RAP**.
2. On cherche toujours des groupes de lettres, on s'autorise maintenant plusieurs options. Par exemple « **[CT]** » signifie « **C** ou **T** ». Ainsi le groupe de lettres « **[CT]O** » correspond aux groupes de lettres « **CO** » ou « **TO** ». Ce groupe est donc contenu dans **TOTEM**, **COTE**, **TOCARD** mais pas dans **VOTER**. De même « **[ABC]** » désignerait « **A** ou **B** ou **C** ».

Nous utiliserons les expressions rationnelles à travers la commande :
`python_regex_chercher(chaine,exp)`

```
from re import *

def python_regex_chercher(chaine,exp):
    trouve = search(exp,chaine)
    if trouve:
        return trouve.group(), trouve.start(), trouve.end()
    else:
        return None
```

Programme-la et teste-la. Elle renvoie : (1) la sous-chaîne trouvée, (2) la position de début et (3) la position de fin.

python : re.search() - python_regex_chercher()

Usage : `search(exp,chaine)`
ou `python_regex_chercher(chaine,exp)`
Entrée : une chaîne de caractères `chaine` et une expression rationnelle `exp`
Sortie : le résultat de la recherche (la sous-chaîne trouvée, sa position de début, celle de fin)

Exemple avec `chaine = "ETRE OU NE PAS ETRE"`

- avec `exp = "P.S"`, alors `python_regex_chercher(chaine,exp)` renvoie `('PAS', 11, 14)`.
- avec `exp = "E..E"`, la fonction renvoie `('ETRE', 0, 4)`.
- avec `exp = "[OT]U"`, la fonction renvoie `('OU', 5, 7)`.
- avec `exp = "[MN]..P[AI]S"`, la fonction renvoie `('NE PAS', 8, 14)`.

Activité 3 (Expressions rationnelles *regex*)**.**

Objectifs : programmer la recherche d'expressions rationnelles simples.

1. Programme ta fonction `regex_chercher_joker(chaine,exp)` qui cherche une sous-chaîne qui peut contenir un ou plusieurs jokers « `.` ». La fonction doit renvoyer : (1) la sous-chaîne trouvée, (2) la position de début et (3) la position de fin (comme pour la fonction `python_regex_chercher()` ci-dessus).
2. Programme ta fonction `regex_chercher_choix(chaine,exp)` qui cherche une sous-chaîne qui peut contenir un ou plusieurs choix contenus dans des balises « `[]` ». La fonction doit de nouveau renvoyer : (1) la sous-chaîne trouvée, (2) la position de début et (3) la position de fin.

   Indication. Tu peux commencer par écrire une fonction `genere_choix(exp)` qui génère toutes les chaînes possibles à partir de `exp`. Par exemple si `exp = "[AB]X[CD]Y"` alors `genere_choix(exp)` renvoie la liste formée de : `"AXCY"`, `"BXCY"`, `"AXDY"` et `"BXDY"`.

Cours 2 (Remplacer des 0 et des 1 et recommencer !)**.**
On considère une « phrase » composée de seulement deux lettres possibles **0** et **1**. Dans cette phrase nous allons chercher un motif (une sous-chaîne) et le remplacer par un autre.

Exemple.
Appliquer la transformation $01 \to 10$ à la phrase 10110.
On lit la phrase de gauche à droite, on trouve le premier motif 01 à partir de la seconde lettre, on le remplace par 10 :

$$1(01)10 \longmapsto 1(10)10$$

On peut recommencer à partir du début de la phrase obtenue, avec toujours la même transformation $01 \to 10$:

$$11(01)0 \longmapsto 11(10)0$$

Le motif 01 n'apparaît plus dans la phrase 11100 donc la transformation $01 \to 10$ laisse maintenant cette phrase inchangée.
Résumons : voici l'effet de la transformation itérée $01 \to 10$ à la phrase 10110 :

$$10110 \longmapsto 11010 \longmapsto 11100$$

Exemple.
Appliquer la transformation $001 \to 1100$ à la phrase 0011.
Une première fois :

$$(001)1 \longmapsto (1100)1$$

Une seconde fois :

$$11(001) \longmapsto 11(1100)$$

Et ensuite la transformation ne modifie plus la phrase.

Exemple.
Voyons un dernier exemple avec la transformation $01 \to 1100$ pour la phrase de départ 0001 :

$$0001 \longmapsto 001100 \longmapsto 01100100 \longmapsto 1100100100 \longmapsto \cdots$$

On peut itérer la transformation, pour obtenir des phrases de plus en plus longues.

Activité 4 (Itérations de remplacements)**.**

Objectifs : étudier quelques transformations et leurs itérations.

On considère ici uniquement des transformations du type $0^a1^b \to 1^c0^d$, c'est-à-dire un motif avec d'abord des 0 puis des 1 est remplacé par un motif avec d'abord des 1 puis des 0.

1. **Une itération.**
   En utilisant ta fonction `remplacer()` de l'activité 1, vérifie les exemples précédents. Vérifie bien que tu ne remplaces qu'un motif à chaque étape (celui le plus à gauche).
   Exemple : la transformation $01 \to 10$ appliquée à la phrase 101, se calcule par `remplacer("101","01","10")` et renvoie `"110"`.
2. **Plusieurs itérations.**
   Programme une fonction `iterations(phrase,motif,nouv_motif)` qui, à partir d'une phrase, itère la transformation. Une fois que la phrase est stabilisée, la fonction renvoie le nombre d'itérations effectuées ainsi que la phrase obtenue. Si le nombre d'itérations n'a pas l'air de s'arrêter (par exemple quand il dépasse 1000) alors renvoie `None`.
   Exemple. Pour la transformation $0011 \to 1100$ et la phrase 000011011, les phrases obtenues sont :

$$000011011 \underset{1}{\longmapsto} 001100011 \underset{2}{\longmapsto} 110000011 \underset{3}{\longmapsto} 110001100 \underset{4}{\longmapsto} 110110000 \longmapsto \text{idem}$$

Pour cet exemple l'appel à la fonction `iterations()` renvoie alors 4 (le nombre de transformations avant stabilisation) et `"110110000"` (la phrase stabilisée).

3. **Le plus d'itérations possibles.**

   Programme une fonction `iteration_maximale(p,motif,nouv_motif)` qui, parmi toutes les phrases de longueur p, cherche l'une de celles qui met le plus de temps à se stabiliser. Cette fonction renvoie :
   - le nombre maximum d'itérations,
   - la première phrase qui réalise ce maximum,
   - la phrase stabilisée correspondante.

   Exemple : pour la transformation **01** → **100**, parmi toutes les phrases de longueur $p = 4$, le maximum d'itérations possibles est 7. Un tel exemple de phrase est **0111**, qui va se stabiliser (après 7 itérations donc) en **11100000000**. Ainsi la commande `iteration_maximale(4,"01","100")` renvoie :

   7, '0111', '11100000000'

   Indication. Pour générer toutes les phrases de longueur p formées de **0** et **1**, tu peux consulter la fiche « Binaire II » (activité 3).

4. **Catégories de transformations.**
   - **Transformation linéaire.** Vérifie expérimentalement que la transformation **0011** → **110** est *linéaire*, c'est-à-dire que pour toutes les phrases de longueur p, il y aura au plus de l'ordre de p itérations au maximum. Par exemple pour $p = 10$, quel est le nombre maximum d'itérations ?
   - **Transformation quadratique.** Vérifie expérimentalement que la transformation **01** → **10** est *quadratique*, c'est-à-dire que pour toutes les phrases de longueur p, il y aura au plus de l'ordre de p^2 itérations au maximum. Par exemple pour $p = 10$, quel est le nombre maximum d'itérations ?
   - **Transformation exponentielle.** Vérifie expérimentalement que la transformation **01** → **110** est *exponentielle*, c'est-à-dire que pour toutes les phrases de longueur p, il y aura un nombre fini d'itérations, mais que ce nombre peut être très grand (beaucoup plus grand que p^2) avant stabilisation. Par exemple pour $p = 10$, quel est le nombre maximum d'itérations ?
   - **Transformation sans fin.** Vérifie expérimentalement que pour la transformation **01** → **1100**, il existe des phrases qui ne vont jamais se stabiliser.