Recherche textuelle

Rappel (chaînes de caractères) Une chaîne de caractères en Python peut être écrite au choix entre apostrophes (caractère `'`) ou entre guillemets (caractère `"`). Ainsi on peut écrire indifféremment `'abracadabra'` ou `"abracadabra"`.
La longueur d'une chaîne de caractères `s` est obtenue avec `len(s)`. Les caractères sont numérotés à partir de 0. Le `(i+1)`e caractère est obtenu avec `s[i]` pour $0 \leq$ `i` $<$ `len(s)`. Les caractères et les chaînes de caractères peuvent être comparés avec l'opérateur `==`.

Exercice 1
Écrire une fonction `affiche` qui prend en paramètre une chaîne de caractères `mot` et affiche chaque caractère de `mot` dans une nouvelle ligne.
Exemple :

```
>>> affiche("NSI")
N
S
I
```

Exercice 2
Écrivez une fonction `longueur`, sans utiliser la fonction `len`, qui prend en paramètre une chaîne de caractères `mot` et qui retourne le nombre de caractères de `mot`.
Test :

```
assert longueur("abracadabra") == 11
```

Exercice 3
Écrire une fonction `recherche` qui prend en paramètres `caractere`, un caractère, et `mot`, une chaîne de caractères, et qui retourne le nombre d'occurrences de `caractere` dans mot, c'est-à-dire le nombre de fois où `caractere` apparaît dans `mot`.
Tests :

```
assert recherche('e', "sciences") == 2
assert recherche('i',"mississippi") == 4
assert recherche('a',"mississippi") == 0
```

Exercice 4
On cherche les occurrences des caractères dans une phrase. On souhaite stocker ces occurrences dans un dictionnaire dont les clefs seraient les caractères de la phrase et les valeurs l'occurrence de ces caractères.
Par exemple : avec la phrase `'Hello world !'` le dictionnaire est le suivant (l'ordre des clefs n'ayant pas d'importance) :
`{'H': 1,'e': 1,'l': 3,'o': 2,' ': 2,'w': 1,'r': 1,'d': 1,'!': 1}`
Écrire une fonction `occurrence_lettres` prenant comme paramètre une variable `phrase` de type `str`. Cette fonction doit retourner un dictionnaire constitué des occurrences des caractères présents dans la phrase.
Utiliser cette fonction pour écrire une fonction `anagramme` qui prend en paramètres deux chaînes de caractères `mot1` et `mot2` et retourne `True` si `mot1` et `mot2` sont des anagrammes et `False` sinon.

Exercice 5

Écrire une fonction `occurrence_max` prenant en paramètres une chaîne de caractères `chaine` et qui retourne le caractère le plus fréquent de la chaîne.
Test :

```
assert occurrence_max("je suis en terminale et je passe le bac") == 'e'
```

Faire en sorte que cette fonction ait la meilleure complexité possible. On pourra par exemple s'inspirer du code de la fonction `occurrence_lettres` précédente.

Exercice 6

Un mot palindrome peut se lire de la même façon de gauche à droite ou de droite à gauche : *bob*, *radar*, et *non* sont des mots palindromes.
l'objectif de cet exercice est d'obtenir un programme Python permettant de tester si une chaîne de caractères est un palindrome.
La fonction `inverse_chaine` inverse l'ordre des caractères d'une chaîne de caractères `chaine` et retourne la chaîne inversée.
La fonction `est_palindrome` teste si une chaîne de caractères `chaine` est un palindrome. Elle retourne `True` si c'est le cas et `False` sinon. Cette fonction s'appuie sur la fonction précédente.
Compléter le code des deux fonctions ci-dessous.

```
def inverse_chaine(chaine):
    result = ...
    for caractere in chaine:
        result = ...
    return result

def est_palindrome(chaine):
    inverse = inverse_chaine(chaine)
    return ...
```

Tests :

```
assert inverse_chaine('bac') == 'cab'
assert not est_palindrome('NSI')
assert est_palindrome('ISN-NSI')
```

Exercice 7

On affecte à chaque lettre de l'alphabet un code selon le tableau ci-dessous :

A	B	C	D	E	F	G	H	I	J	K	L	M
1	2	3	4	5	6	7	8	9	10	11	12	13
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
14	15	16	17	18	19	20	21	22	23	24	25	26

Pour un mot donné, on détermine d'une part son *code alphabétique concaténé*, obtenu par la juxtaposition des codes de chacun de ses caractères, et d'autre part, son *code additionné*, qui est la somme des codes de chacun de ses caractères.
Par ailleurs, on dit que ce mot est « parfait » si le code additionné divise le code concaténé.
Exemples :

- Pour le mot `"PAUL"`, le code concaténé est la chaîne `"1612112"`, soit l'entier 1 612 112.
  Son code additionné est l'entier 50 car $16 + 1 + 21 + 12 = 50$. 50 ne divise pas l'entier 1 612 112 ; par conséquent, le mot `"PAUL"` n'est pas parfait.

- Pour le mot "ALAIN", le code concaténé est la chaîne "1121914", soit l'entier 1 121 914. Le code additionné est l'entier 37 car $1 + 12 + 1 + 9 + 14 = 37$. 37 divise l'entier 1 121 914 ; par conséquent, le mot "ALAIN" est parfait.

Compléter la fonction `est_parfait` ci-dessous qui prend comme argument une chaîne de caractères `mot` (en lettres majuscules) et qui retourne son code additionné, son code concaténé (sous forme d'un entier), ainsi qu'un booléen qui indique si `mot` est parfait ou pas.

```
dico = {chr(ord('A')+i):i+1 for i in range(26)}

def est_parfait(mot) :
    code_a = ...
    code_c = ""
    for c in mot :
        code_a = ...
        code_c = code_c + ...
    code_c = int(code_c)
    if ... :
        mot_est_parfait = True
    else:
        mot_est_parfait = False
    return code_a, code_c, mot_est_parfait
```

Tests :

```
assert est_parfait("PAUL") == (50, 1612112, False)
assert est_parfait("ALAIN") == (37, 1121914, True)
```

Exercice 8

La fonction `recherche` prend en paramètres deux chaînes de caractères `gene` et `seq_adn` et retourne `True` si on retrouve `gene` dans `seq_adn` et `False` sinon.

Compléter le code Python ci-dessous pour qu'il implémente la fonction recherche.

```
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    i = ...
    trouve = False
    while i < ... and ... :
        j = 0
        while j < g and gene[j] == seq_adn[i+j]:
            ...
            if j == g:
                trouve = True
        ...
    return trouve
```

Tests :

```
assert recherche("AATC", "GTACAAATCTTGCC")
assert not recherche("AGTC", "GTACAAATCTTGCC")
```