Semaine 9 : mercredi 11/12 au mercredi 18/12

Exercice 1
Cet exercice utilise le type `list` de *Python* pour implémenter une liste. Ainsi on utilisera uniquement les syntaxes suivantes :

- pour créer une pile vide : `ma_pile = []`
- pour vérifier que la pile est vide : l'expression `len(ma_pile) == 0` est évaluée en `True` lorsque la pile est vide
- pour empiler un élément dans une pile on utilise la méthode `append` : `ma_pile.append(elt)`
- pour dépiler une pile en renvoyant la valeur dépiler on utilise la méthode `pop` : `elt = ma_pile.pop()`

Écrire deux fonctions `pile_positive_v.`, prenant en paramètre une pile `pile` de nombres entiers et qui renvoie une pile contenant uniquement les valeurs positives de la pile initiale `pile`. De plus, la pile initiale doit retrouver son état d'origine à la fin de l'exécution de la fonction.
La version 1 renvoie une pile dont le premier nombre positif depuis le sommet de la pile d'origine se retrouve au sommet de la pile renvoyée. Et ainsi de suite pour les autres nombres positifs.
La version 2 renvoie une pile dont le premier nombre positif depuis le sommet de la pile d'origine se retrouve au fond de la pile renvoyée. Et ainsi de suite pour les autres nombres positifs.

Exemples :

```
>>> ma_pile = [-1, 0, 5, -3, 4, -6, 10, 9, -8]
>>> pile_positive_v1(ma_pile)
[5, 4, 10, 9]
>>> pile_positive_v2(ma_pile)
[9, 10, 4, 5]
>>> ma_pile
[-1, 0, 5, -3, 4, -6, 10, 9, -8]
>>> ma_pile = [-2, -3, 0, -4]
>>> pile_positive_v1(ma_pile)
[]
>>> pile_positive_v2(ma_pile)
[]
>>> ma_pile
[-2, -3, 0, -4]
>>> ma_pile = [8, 5, 1, 3]
>>> pile_positive_v1(ma_pile)
[8, 5, 1, 3]
>>> pile_positive_v2(ma_pile)
[3, 1, 5, 8]
>>> ma_pile
[8, 5, 1, 3]
```

Exercice 2 :
On possède la classe suivante pour effectuer l'implémentation d'une pile :

```
class Pile:
    """Classe implémentant la structure de données d'une pile."""
    def __init__(self):
        self.contenu = []

    def est_vide(self):
        """
        Renvoie le booléen True si la pile est vide, False sinon.
        """
        return self.contenu == []

    def empiler(self, v):
        """
        Place l'élément v au sommet de la pile.
```

```
        """
        self.contenu.append(v)

    def depiler(self):
        """
        Retire et renvoie l'élément placé au sommet de la pile,
        si la pile n'est pas vide.
        """
        if not self.est_vide():
            return self.contenu.pop()
```

Une expression arithmétique peut utiliser des parenthèses pour préciser les priorités de calculs. Parfois, une expression a besoin de plusieurs paires de parenthèses et pour faciliter la lecture on peut utiliser des crochets ou des accolades.
Pour simplifier l'expression étudiée, on étudiera uniquement des chaînes de caractères utilisant les symboles `'('`, `')'`, `'['`, `']'`, `'{'` ou `'}'`, sans tenir compte des nombres et des opérations.
On dit qu'une expression est correctement balisée lorsque chaque balise ouvrante possède sa balise fermante correspondante. De plus, avant que cette balise fermante soit fermée, il faut que les balises entre les deux forment une expression correctement balisée.

Par exemples :

- `'({[}])'` n'est pas correctement balisée comme on ferme l'accolade avant d'avoir fermé le crochet
- `'[({}[])(]'` n'est pas correctement balisée comme la dernière parenthèse ouvrante ne possède pas de parenthèse fermante
- `'{[()[{}]()]({}[])}'` est correctement balisée

Écrire une fonction `bon_balisage`, prenant en paramètre une chaîne de caractère `expression` et qui renvoie `True` lorsque cette expression est correctement balisée, et `False` sinon.

Par exemples :

```
>>> bon_balisage('{[()[{}]()]({}[])}')
True
>>> bon_balisage('({[}])')
False
>>> bon_balisage('[({}[])(]')
False
```

Pour éviter ces problèmes de balisages incorrectes, on utilise la notation polonaise inverse pour écrire des expressions arithmétiques.
Pour simplifier l'expression étudiée, on utilisera uniquement les opérations d'addition et de multiplication.
Par exemples :

- l'expression arithmétique $(2+3)\times 5$ va s'écrire `'2 3 + 5 *'`
- l'expression arithmétique $3\times 2+5$ va s'écrire `'3 2 * 5 +'`
- l'expression arithmétique $2+5\times 3$ va s'écrire `'2 5 3 * +'`

D'une manière plus générale, l'évaluation d'une expression arithmétique en notation polonaise inverse est déterminée à l'aide d'une pile en parcourant l'expression arithmétique de gauche à droite de la façon suivante :

- si l'élément parcouru est un nombre, on le place au sommet de la pile ;
- si l'élément parcouru est un opérateur, on récupère les deux éléments situés au sommet de la pile et on leur applique l'opérateur. On place alors le résultat au sommet de la pile.

À la fin du parcours, il reste alors un seul élément dans la pile qui est le résultat de l'expression arithmétique.

On pourra utiliser la méthode `'split'` sur une chaîne de caractères qui permet d'obtenir une liste de chaînes de caractères obtenu en séparant la chaîne de caractères suivant le caractère précisé. Dans notre situation, on veut séparer notre chaîne de caractères grâce aux espaces.
Par exemple :

```
>>> '2 3 + 5 *'.split(' ')
['2', '3', '+', '5', '*']
```

Écrire une fonction `evaluation`, prenant en paramètre une chaîne de caractère `expression` et qui renvoie le résultat de l'expression écrite en notation polonaise inverse.

Par exemples :

```
>>> evaluation('2 3 + 5 *')
25
>>> evaluation('3 2 * 5 +')
11
>>> evaluation('2 5 3 * +')
17
```

Exercice 3 :
L'opérateur « ou exclusif » entre deux bits renvoie 0 si les deux bits sont égaux et 1 s'ils sont différents :

A	B	A oux B
0	0	0
0	1	1
1	0	1
1	1	0

On représente ici une suite de bits par un tableau contenant des `0` et des `1`.
Exemples :

```
bits1 = [1, 1, 0, 0, 1, 0, 0, 1]
bits2 = [0, 1, 1, 0, 0, 1, 1, 1]
bits3 = [1, 0, 0, 0]
bits4 = [1, 1, 0, 1]
```

Écrire une fonction `oux_bit_par_bit`, prenant en paramètres deux tableaux de même longueur `tab1` et `tab2` et qui renvoie un tableau de même longueur où l'élément situé à l'indice `i` est le résultat, par l'opérateur « ou exclusif », des éléments aux indices `i` des tableaux passés en paramètres.
On ajoutera une assertion permettant de vérifier que les tableaux passés en paramètres sont de même longueurs.

Exemples :

```
>>> oux_bit_par_bit(bits1, bits2)
[1, 0, 1, 0, 1, 1, 1, 0]
>>> oux_bit_par_bit(bits3, bits4)
[0, 1, 0, 1]
>>> oux_bit_par_bit(bits1, bits3)
Traceback (most recent call last):
AssertionError: les tableaux doivent être de même longueur
```

Exercice 4 :
Le chiffrement de César transforme un message en changeant chaque lettre en la décalant dans l'alphabet.
Par exemple, avec un décalage de 3, le A se transforme en D, le B en E, ..., le X en A, le Y en B et le Z en C. Les autres caractères ne sont pas chiffrés.

Pour effectuer le décalage on associera un numéro à chaque lettre à l'aide du dictionnaire suivant :

```
NUM_LETTRE = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9,\
              'K': 10, 'L': 11, 'M': 12, 'N': 13, 'O': 14, 'P': 15, 'Q': 16, 'R': 17,\
              'S': 18, 'T': 19, 'U': 20, 'V': 21, 'W': 22, 'X': 23, 'Y': 24, 'Z': 25}
```

Pour effectuer l'action réciproque on utilisera la chaîne de caractères suivante :

```
ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

Ainsi :

```
>>> NUM_LETTRE['R']
17
>>> ALPHABET[17]
'R'
```

Écrire une fonction `cesar`, prenant en paramètres une chaîne de caractères `message` et un nombre entier `decalage` et qui renvoie le nouveau message chiffré avec le chiffrement de César utilisant le décalage passé en paramètre.

Exemples :

```
>>> cesar('BONJOUR A TOUS. VIVE LA CRYPTOGRAPHIE !', 6)
'HUTPUAX G ZUAY. BOBK RG IXEVZUMXGVNOK !'
>>> cesar('HUTPUAX G ZUAY. BOBK RG IXEVZUMXGVNOK !', -6)
'BONJOUR A TOUS. VIVE LA CRYPTOGRAPHIE !'
```

Le chiffrement de Vigenère améliore le chiffrement de César en appliquant un décalage qui n'est pas identique pour tout le message.
Par exemple, si le message est `EXEMPLE` et si on utilise le code `NSI`, alors :

- le code commence par `N`, donc on commence avec un décalage de 13 pour le `E`
- ensuite on a `S` dans le code, donc on poursuit avec un décalage de 18 pour le `X`
- ensuite on a `I` dans le code, donc on poursuit avec un décalage de 8 pour le `E`
- le code étant fini, on repart avec `S`, donc on poursuit avec un décalage de 13 pour le `M`
- etc ...

On conserve le principe que les autres caractères que les lettres capitales ne sont pas chiffrés.

Écrire une fonction `vigenere`, prenant en paramètres deux chaînes de caractères `message` et `code` et qui renvoie le nouveau message chiffré avec le chiffrement de Vigenère utilisant le mot code passé en paramètre.

Exemples :

```
>>> vigenere('BONJOUR A TOUS. VIVE LA CRYPTOGRAPHIE !', 'FELICITATION')
'GSYRQCK T HBZW. XQOE TO HVJXVWZRTXVVJ !'
>>> vigenere('GSYRQCK T HBZW. XQOE TO HVJXVWZRTXVVJ !', 'VWPSYSHAHSMN')
'BONJOUR A TOUS. VIVE LA CRYPTOGRAPHIE !'
```