Séance 03 :

Piles et Files

1. Création d'une pile

1. ***Créer un module appelé*** `Pile` ***qui contient les fonctions suivantes :***
   - `creer_pile` ***qui crée une pile vide sous la forme d'une liste vide***
   - `empiler` ***qui prend en argument une pile*** `p` ***et un élément*** `e` ***et l'empile sur la pile***
   - `sommet` ***qui prend en argument une pile*** `p` ***et renvoie son sommet sans modifier la pile***
   - `depiler` ***qui prend en argument une pile*** `p` ***et dépile son sommet***
   - `hauteur` ***qui prend en argument une pile*** `p` ***et renvoie le nombre d'éléments de cette pile***
   - `pile_vide` ***qui prend en argument une pile*** `p` ***et renvoie*** `True` ***si elle est vide,*** `False` ***sinon***

   Pour des raisons évidentes, je choisis de créer une classe `Pile` et de transformer toutes ces fonctions en méthodes. Pour plus de classe, je renomme ces méthodes en anglais.

```python
class Pile:
    _pile = []

    def __init__(self):
        self._pile = []

    def stack(self, element):
        """Add element at the end of the pile"""
        self._pile.append(element)

    def top(self):
        """Return the last element of the pile without modifying it"""
        return self._pile[-1:]

    def unstack(self):
        """Return the last element of the pile, and remove it from the pile.
    """
        return self._pile.pop()

    def lenght(self):
        """Return lenght of the pile."""
        return len(self._pile)

    def isEmpty(self):
        """Return true if pile is empty."""
        return len(self._pile) == 0

    def toString(self):
        """Return pile in a str type"""
        return str(self._pile)
```

2. Vérification du parenthésage d'une expression

On considère des expressions algébriques données sous forme de chaîne :

```
c1 = '((3*5-7)+1'
c2 = '((3*5)-7)+1'
c3 = '(3*5)-(7)+1)'
c4 = '(3*(5-7)+(1'
```

On souhaite faire une analyse synthaxique pour vérifier qu'à chaque parenthèse ouvrante correspond une unique parentèse fermante.

1. ***Proposer un algorithme utilisant une pile et réalisant ce test.***
   On crée un pile vide.
   A chaque parenthèse ouvrante, on empile 1.
   A chaque parenthèse fermante, on dépile 1.
   A la fin, si la pile n'est pas vide, c'est qu'il y a une erreur dans l'expression algébrique.
2. ***Ecrire une fonction*** `test_parentheses(c)` ***qui implémente cet algorithme.***

```
def parenthesisTest(chaine):
    pile = Pile()
    for char in chaine:
        if char == '(':
            pile.stack(1)
        elif char == ')':
            if not pile.isEmpty():
                pile.unstack()
            else:
                return False
    if pile.isEmpty():
        return True
    else:
        return False
```

3. Décimal - Binaire

L'objectif est de créer un programme qui code un nombre entier décimal en binaire. Ainsi, 255 ***en décimal correspond à*** 1000001101 ***en binaire.***

1. ***Ecrire une fonction*** `div2(x)` ***qui retourne la division entière par 2 d'un nombre entier et le reste de cette division.***

```
def div2(x):
    return (x//2, x % 2)
```

2. ***Ecrire le programme qui met dans une pile les caractères représentant l'image binaire inversée du nombre décimal.***

```
def stackBinary(decimalNumber):
    pile = Pile()
    while decimalNumber >= 1:
        a = div2(decimalNumber)
        decimalNumber = a[0]
        pile.stack(a[1])
    return pile
```

3. ***Faire afficher ce résultat.***

```
print(stackBinary(256).toString())
```

4. ***Pour plus de lisibilité, un espace sera inséré tous les 4 caractères en partant du dernier. Modifier le programme précédent pour intégrer cette nouvelle exigence.***

```
def displayIt(pile):
    a, i = '', 0
    iMax = pile.lenght() % 4
    while not pile.isEmpty():
        if (i == iMax):
            a += ' '
            iMax, i = 4, 0
        a += str(pile.unstack())
        i += 1
    return a
```

4. Notation Polonaise inverse

1. *Créer une fonction* `est_operateur` *qui prend une chaîne de caractère en argument et renvoie* `True` *si cette chaîne est* `"+"`, `"*"`, `"/"` *ou* `"-"`, `False` *sinon*

```
def isOperator(char):
    op = [
        '+',
        '-'
        '/',
        '*'
    ]
    return char in op
```

2. *Créer une fonction* `calcul` *qui prend en argument un opérateut* `op` *et deux opérandes* `a` *et* `b` *et qui renvoie la valeur de* `a op b`.

```
def calculOp(a, b, op):
    if op == '+':
        return a + b
    elif op == '-':
        return a - b
    elif op == '/':
        return a / b
    elif op == '*':
        return a * b
    else:
        raise '{} ne fait pas partie de la liste des operateurs'.format(op)
```

3. *Définir une fonction* `liste_mots` *qui prend en argument une chaîne de caractères et retourne la liste des caractères non séparés par un espace.*

```
def listWords(chaine):
    return [a for a in chaine.replace(' ', '')]
```

4. *En utilisant les deux fonctions précédentes, définir la fonction* `polonaise`.

```
def polonaise(chaine):
    pile = Pile()
    for char in listWords(chaine):
        if isOperator(char):
            if pile.lenght() >= 2:
                a = pile.unstack()
                b = pile.unstack()
                pile.stack(calculOp(a, b, char))
            else:
                raise 'Error ! (oups)'
        elif char.isnumeric():
            pile.stack(float(char))
        else:
            raise 'A non numeric character has been found ({})'.format(char)
    if pile.lenght() == 1:
        return pile.unstack()
    else:
        raise 'Error (oups) !'
```

5. Création d'une file

1. *Créer un module appelé* `File` *qui contient les fonctions suivantes :*
   - `creer_file` *qui crée une file vide sous la forme d'une liste vide*
   - `push` *qui prend en argument une file* `f` *et un élément* `e` *et l'insère dans la file*

— `pop` ***qui prend en argument une file*** `f` ***et enlève et renvoie l'élément en tête de file***

— `longueur` ***qui prend en argument une file*** `f` ***et renvoie le nombre d'éléments de cette file***

— `file_vide` ***qui prend en argument une file*** `f` ***et renvoie*** `True` ***si elle est vide,*** `False` ***sinon***

De même, je choisis de créer une classe `File`, de transformer toutes ces fonctions en méthodes et je renomme ces méthodes en anglais.

```
class File:
    _file = []

    def __init__(self):
        self._file = []

    def push(self, element):
        """Add element at the end of the file"""
        self._file.append(element)

    def top(self):
        """Return the first element of the file without modifying it"""
        return self._file[:1]

    def pop(self):
        """Return the first element of the file, and remove it from the file
    ."""
        a = self._file[:1]
        self._file = self._file[1:]
        return a

    def lenght(self):
        """Return lenght of the file."""
        return len(self._file)

    def isEmpty(self):
        """Return true if file is empty."""
        return len(self._file) == 0

    def toString(self):
        """Return file in a str type"""
        return str(self._file)
```