Exercice 1 – Construction d'une pile

Correction **Question 1**

```
def creer_pile(n):
    """
    Créer une pile de taille n.
    Entrée :
     * n(int) : taille souhaitée de la pile
    Sortie :
     * pile(list) : pile de taille n.
     sinon
    """
    return n*[None]
```

Question 2

```
def est_vide(pile):
    """
    Vérifie si la pile est vide.
    Entrée :
     * pile(list)
    Sortie :
     * retourne True si la pile est vide, False
     sinon
    """
    for el in pile :
        if el != None :
            return False
    return True
```

Question 3

```
def est_pleine(pile):
    """
    Vérifie si la pile est pleine.
    Entrée :
     * pile(list)
     * nb(int) : nombre d'éléments maximum dans une pile
    Sortie :
     * retourne True si la pile est pleine, False
     sinon
    """
    return (not None in pile)
```

Question 4

```
def empiler(pile,el):
    """
    Empile l'élément el sur la pile pile.
    Entrée :
     * pile(list)
     * el(*) : objet àempiler
    Sortie :
     * ne fait rien si la pile est déjà pleine
     * retourne None dans tous les cas
    """
    if est_pleine(pile):
        return None
    # On recherche le premier emplacement vide
    i=0
    while pile[i]!= None:
        i=i+1
    pile[i]=el
```

Question 5

```
def depiler(pile):
    """
    Dépile l'élément du haut de la pile pile.
    Entrée :
     * pile(list)
    Sortie :
     * None si la pile est vide
     * l'élément du haut de la pile sinon.
     """
    if est_vide(pile):
        return None
    i=0
    # On recherche le premier emplacement vide
    while i < len(pile) and pile[i] != None :
        i=i+1
    el = pile[i-1]
    pile[i-1]=None
    return el
```

Question 6

```
def taille_pile(pile):
    """
    Renvoie la taille de la pile pile (pas le nombre d'éléments empilés !.
    Entrée :
     * pile(list)
    Sortie :
     * un entier naturel
    """
    return len(pile)
```

Exercice 2

Correction

```
def concat_it (P1,P2) :
    """ concatène deux piles en plaçant P1 "sur" P2
    de manière itérative.
    Préconditon : P2 est assez profonde pour contenir P1 """
    n = taille_pile (P1)
    P3 = creer_pile(n) # on recopie P1 àl'envers dans P3
    while est_vide(P1) == False :
        x = depiler(P1)
        empiler(P3,x)
    while est_vide(P3) == False : # on rempile P3 àl'envers dans P2
        x = depiler(P3)
        empiler(P2,x)

def concat_rec (P1,P2) :
    """ concatène deux piles en plaçant P1 "sur" P2
    de manière récursive.
    Préconditon : P2 est assez profonde pour contenir P1 """
    if est_vide(P1) or est_pleine(P2):
        return None
    else :
        x = depiler(P1)
        concat_rec(P1,P2)
        empiler(P2,x)
```

Exercice 3

Correction

```
def somme3 (P) :
    """ Renvoie la plus grande somme de trois éléments consécutifs
    de la pile P.
    Si cette pile a deux éléments ou moins, renvoie 0.
    Précondition : P est une pile d'entiers naturels """
    R = P.copy()
```

```
    if est_vide(R) :
        return 0
    else :
        x = depiler(R)
        s = somme3(R)
        for i in range(2) :
            if est_vide(R) :
                return s
            else :
                x += depiler(R)
        return max(s,x)
```

Exercice 4

Correction

```
def copie (P,n) :
    """ recopie la pile P dans une pile de taille n.
    préconditon : n > taille_pile(P) """
    l = taille_pile(P)
    R = P.copy()
    assert n>l
    if est_vide(R) :
        return creer_pile(n)
    else :
        x = depiler(R)
        S=copie(R,n)
        empiler(S,x)
        return S

def fusion (P1,P2) :
    """ Fusionne les deux piles P1 et P2 en respectant
    les règles de l'énoncé """
    n = taille_pile(P1)+taille_pile(P2)
    R1 = P1.copy()
    R2 = P2.copy()
    if est_vide(R1) :
        return copie(R2,n)
    if est_vide(R2) :
        return copie(R1,n)
    else :
        e1 = depiler(R1)
        e2 = depiler(R2)
        R3 = fusion(R1,R2)
        empiler(R3,e1)
        empiler(R3,e2)
        return R3
```

Exercice 5 – Notation polonaise inversée

Correction **Question 1**

```
def est_nombre(el):
    return type(el)==float or type(el)==int
```

Question 2

```
def est_operation(el):
    return el in ["+","-","*","/"]
```

Question 3

```
def inversion(pile):
    pile2=creer_pile(taille_pile(pile))
    pile3=creer_pile(taille_pile(pile))
    while not (est_vide(pile)):
        empiler(pile2,depiler(pile))
    while not (est_vide(pile2)):
        empiler(pile3,depiler(pile2))
    while not (est_vide(pile3)):
```

```
        empiler(pile,depiler(pile3))
```

Question 4

```
def operer(nb1,nb2,op):
    if op == "+":
        return nb1+nb2
    elif op == "*":
        return nb1*nb2

def evaluer(pile):
    """
    Évaluer le résultat d'une opération post-fixée.
    Entrée :
     * pile(lst) : liste d'opérateurs et d'opérandes
    Sortie :
     * res(flt) : résultat du calcul de l'expression.
    """
    inversion(pile)
    pile2=creer_pile(taille_pile(pile))
    while not est_vide(pile):
        el = depiler(pile)
        if est_nombre(el):
            empiler(pile2,el)
        elif est_operation(el) :
            empiler(pile2,operer(depiler(pile2),depiler(pile2),el))
    return depiler(pile2)
```

Question 5

```
pile1=[1,2,"+",4,"*",3,"+"]
pile2=[1,2,"+",4,"*",-3,"+",5,"+"]
print(evaluer(pile2))
```