Planche 1

On s'intéresse à l'analyse et à l'évaluation d'expressions arithmétiques représentées sous forme de chaînes de caractères ou de listes.

Rappels sur les chaînes de caractères : les chaînes se manipulent comme des listes, en particulier :

`chaine = ""` :	chaîne vide
`chaine[i]` :	caractère de rang i dans `chaine`
`len(ch)` :	longueur de la chaîne
`car in chaine` :	teste si le caractère `car` apparaît dans `chaine`
`ch[deb:fin]` :	extrait la sous-chaîne entre les indices `deb` (inclus) et `fin` (exclu)
`int(x)` :	convertit la chaîne `x` en entier : `int("367")` → `367`
`x.isdigit()` :	renvoie `True` si `x` n'est constitué que de chiffres, `False` sinon

1. **Somme**

   On considère des expressions sous forme de chaînes de caractères décrivant des sommes d'entiers naturels. Ces chaînes ne comprennent que des **chiffres** et le **symbole** `'+'`.

   Exemple d'expression : `"123+15+1"`

   Écrire une fonction `est_valide(ch)` qui renvoie `True` si la chaîne est valide, c'est-à-dire :

   - qu'elle ne contient que des caractères autorisés (chiffres et `'+'`) ;
   - que les opérateurs `'+'` sont bien placés : ni au début, ni à la fin, ni en doublon.

   La fonction devra renvoyer `False` sinon.

   Expressions bien formées : `"123"`, `"123+5"`, `"123+15"`, `"123+15+1"`

   Expressions mal formées : `""`, `"12+"`, `"+12"`, `"12++12"`, `"12+x+5"`

2. **Évaluation**

   Écrire une fonction `evalue(ch)` qui calcule et renvoie la valeur d'une expression bien formée passée en paramètre.

   Exemple : `evalue("123+15+1")` renvoie `139`.

   Indication : la méthode `ch.split(car)` renvoie la liste des sous-chaînes de `ch` obtenues en découpant selon le séparateur `car`.

   Exemple : si `ch = "Hydrogene-1.0079-H-1"`, alors `ch.split("-")` renvoie `["Hydrogene", "1.0079", "H", "1"]`.

3. **Liste d'additions**

   On dispose d'une liste d'expressions arithmétiques telles que décrites précédemment. Écrire un programme qui calcule et affiche la valeur la plus grande parmi ces expressions.

   Exemple : pour la liste `["123+15", "123+15+1", "123", "123+5"]`, le programme devra renvoyer 139.

4. **Expressions arithmétiques générales**

   On souhaite généraliser en ajoutant les opérateurs `'-'`, `'*'` et `'/'`.

   Écrire une fonction `calcule(a, b, op)` qui reçoit deux opérandes et un opérateur et renvoie le résultat de l'opération.

   Exemple : `calcule(546, 23, '-')` renvoie `523`.

5. **Évaluation sans priorité**

   On suppose qu'une expression arithmétique est représentée par une liste alternant opérandes et opérateurs.

   Exemple : `[46, '+', 20, '-', 10, '*', 2]`

Écrire une fonction `evaluation(L)` qui calcule la valeur correspondante en effectuant les opérations de gauche à droite, sans tenir compte des priorités.

Exemple : `[46, '+', 20, '−', 10, '∗', 2]` est évalué comme $(46 + 20 - 10) \times 2 = 112$.

On utilisera la fonction `calcule` de la question précédente.

6. **Notation postfixée**

Pour respecter la priorité des opérateurs, on utilise la *notation postfixée* (notation polonaise inverse) : l'opérateur est placé *après* ses deux opérandes.

Exemple : la liste `[46, 20, '+', 10, 2, '∗', '−']` correspond à $46 + 20 - 10 \times 2$.

L'algorithme d'évaluation utilise une pile :

- on empile les opérandes rencontrés successivement ;
- lorsqu'on rencontre un opérateur, on dépile les deux derniers entiers, on leur applique l'opérateur, et on empile le résultat ;
- on procède ainsi jusqu'à épuisement de la liste.

Écrire une fonction qui calcule et renvoie le résultat d'une expression arithmétique postfixée représentée sous forme de liste.

7. **Vérification d'erreurs**

Compléter la fonction précédente pour qu'elle détecte les erreurs suivantes :

- une division par zéro ;
- un nombre d'opérateurs trop grand ou insuffisant.

Dans ces cas, la fonction renvoie `"ERR"`.

Corrigé — Planche 1

1. **Somme**

```
def est_valide(ch):
    if not (ch[0].isdigit() and ch[-1].isdigit()):
        return False
    nombre_avant_plus = True
    for x in ch[1:]:
        if x == "+":
            if not nombre_avant_plus:
                return False
            nombre_avant_plus = False
        elif x.isdigit():
            nombre_avant_plus = True
        else:
            return False
    return True
```

2. **Évaluation**

```
def evalue(ch):
    L = ch.split("+")
    return sum([int(x) for x in L])
```

3. **Liste d'additions**

```
def plus_grande_valeur(L):
    v_max = evalue(L[0])
    for x in L[1:]:
        v = evalue(x)
        if v > v_max:
            v_max = v
    return v_max
```

4. **Expressions arithmétiques générales**

```
def calcule(a,b,op):
    if op == "+":
        return a+b
    elif op == "-":
        return a-b
    elif op == "*":
        return a*b
    elif op == "/":
        return a/b
```

5. **Évaluation sans priorité**

```
def evalue_ss_priorite(L):
    s = L[0]
    for k in range(1,len(L),2):
        s = calcule(s,L[k+1],L[k])
    return s
```

6. **Notation postfixée**

```
def evaluation_postfixe(L):
    pile = []
    for x in L:
        if x not in ["+","-","/","*"]:
            pile.append(x)
        else:
            b = pile.pop()
            a = pile.pop()
            pile.append(calcule(a,b,x))
    return pile[0]
```

7. **Vérification d'erreurs**

```
def evaluation_postfixe(L):
    pile = []
    for x in L:
        if x not in ["+","-","/","*"]:
            pile.append(x)
        else:
            if len(pile) < 2:
                return "ERR"
            b = pile.pop()
            a = pile.pop()
            if (b,x) == (0,"/"):
                return "ERR"
            pile.append(calcule(a,b,x))
    if len(pile) == 1:
        return pile[0]
    else:
        return "ERR"
```

Planche 2

On s'intéresse à différentes méthodes de calcul et d'exploration du nombre π.

1. **Formule de Leibniz**

   Une valeur approchée de π peut être calculée grâce à la formule de Leibniz :

   $$\frac{\pi}{4} = \sum_{i=1}^{+\infty} \frac{(-1)^{i+1}}{2i-1}.$$

   a. **Terme.** Écrire une fonction `terme(i)` qui calcule le terme de rang i de cette suite :

   $$t_i = \frac{(-1)^{i+1}}{2i-1}.$$

   b. **Suite.** Écrire une fonction qui calcule la valeur approchée de π par cette formule jusqu'au rang n donné en paramètre.

   (i) En utilisant la fonction écrite à la question précédente.

   (ii) *(Optionnel)* Sans utiliser la fonction précédente. On peut optimiser le nombre d'opérations à chaque étape en mettant en évidence la récurrence entre les termes T_i de la suite :

   $$T_i = \frac{num_i}{den_i}, \qquad num_i = (-1)^{i+1}, \qquad den_i = 2i-1.$$

   c. **Calcul borné.** Dans le module `math` de Python, la constante `m.pi` fournit une valeur de référence de π.

   (i) Modifier la fonction précédente pour que le calcul s'arrête dès que la valeur approchée obtenue diffère de `m.pi` de moins de `epsilon` donné en paramètre.

   (ii) Modifier à nouveau la fonction pour qu'elle renvoie la valeur approchée obtenue **ainsi que** le nombre d'itérations effectuées.

2. **Décimales de π**

   On dispose de la liste des 10 000 premières décimales de π :

   ```
   pi_decimales = [1,4,1,5,9,2,6,5,3,5,8,9,7,9,3,2,3,8,4,6,2,6,4,3,3,8,3,2,7,9,5,0,2,8,8,4,...]
   ```

   a. **Les 20 premières décimales.** Il paraît que la somme des 20 premières décimales de π vaut 100. Écrire les instructions qui permettent de vérifier cette propriété.

   b. **Le chiffre 0.** Écrire une fonction qui renvoie le rang auquel apparaît le premier `0` dans `pi_decimales`, ou `False` si aucun `0` n'y figure.

   c. **Affichage.** Pour faciliter la lecture, on souhaite afficher les décimales sous forme de grille : 10 décimales par ligne, séparées par un espace.

   Exemple :

   ```
   1 4 1 5 9 2 6 5 3 5
   8 9 7 9 3 2 3 8 4 6
   2 6 4 3 3 8 3 2 7 9
   ...
   ```

   Écrire les instructions qui produisent cet affichage.

   Indications :

   - On suppose que la liste contient un nombre de valeurs multiple de 10.
   - On effectuera une double itération : une sur le nombre de lignes, une sur les 10 décimales de chaque ligne.
   - `print(val, end="")` affiche `val` suivi d'une espace, sans passer à la ligne.

d. **Date de naissance.** Écrire une fonction qui recherche si une date de naissance donnée apparaît dans les décimales de π et renvoie le rang à partir duquel elle apparaît.

Exemple : la date `29021960` apparaît à partir du rang 712.

Indication : on commencera par transformer la date en liste de chiffres.

e. **Chiffres.** Écrire une fonction qui renvoie le rang à partir duquel tous les chiffres de 0 à 9 sont apparus au moins une fois dans `pi_decimales`.

f. **Les chiffres encore.** Modifier la fonction de la question 1.c pour que le calcul s'arrête dès que tous les chiffres de 0 à 9 sont apparus au moins une fois parmi les décimales de la valeur approchée calculée.

Indication : on suppose qu'on dispose d'une fonction `decimales` qui fournit la liste des décimales d'un réel ; par exemple, l'instruction `decimales(3,14159)` devra renvoyer `[1,4,1,5,9]`.

Corrigé — Planche 2

1. **Formule de Leibniz**

a.
```
def terme(i):
    return ((-1)**(i+1))/(2*i-1)
```

b. (i)
```
def approx_pi(n):
    s = 0
    for i in range(1,n+1):
        s += terme(i)
    return 4*s
```

(ii)
```
def approx_pi2(n):
    num = 1
    den = 1
    s = num/den
    for i in range(2,n+1):
        num = -num
        den = den + 2
        s += num/den
    return 4*s
```

c. (i)
```
import math as m

def approx_pi_borne(epsilon):
    s = 0
    i = 1
    while abs(m.pi - 4*s) > epsilon:
        s += terme(i)
        i += 1
    return 4*s
```

(ii)
```
import math as m

def approx_pi_borne(epsilon):
    s = 0
    i = 1
    nb_iter = 0
    while abs(m.pi - 4*s) > epsilon:
        s += terme(i)
        i += 1
        nb_iter += 1
    return 4*s, nb_iter
```

2. **Décimales de π**

a.
```
print(sum(pi_decimales[:20])==100)
```

b.
```
def rang0():
    for k in range(len(pi_decimales)):
        if pi_decimales[k] == 0:
            return k
    return False
```

c.
```
def affichage(L):
    for k in range(len(L)//10):
        for i in range(10):
            print(L[10*k+i],end=" ")
        print("")
```

d.
```
def date_naissance(date):
    liste_date = [int(x) for x in str(date)]
    n = len(liste_date)
    for k in range(len(pi_decimales)-n):
        if pi_decimales[k:k+n] == liste_date:
            return k
    return False
```

e.
```
def rang_tous_chiffres():
    chiffre_present = [0 for k in range(10)]
    nb_chiffres_trouves = 0
    for k in range(len(pi_decimales)):
        x = pi_decimales[k]
        if chiffre_present[x] == 0:
            chiffre_present[x] = 1
            nb_chiffres_trouves += 1
            if nb_chiffres_trouves == 10:
                return k
```

f.
```
def dix_decimales_differentes(nb):
    chiffres = [0 for k in range(10)]
    d = decimales(nb)
    nb_chiffres = 0
    for x in nb:
        if chiffres[x] == 0:
            chiffres[x] = 1
            nb_chiffres += 1
    return nb_chiffres == 10


def approx_pi_chiffres(epsilon):
    s = 0
    i = 1
    nb_iter = 0
    while dix_decimales_differentes(4*s):
        s += terme(i)
        i += 1
        nb_iter += 1
    return 4*s, nb_iter
```

Planche 3

On s'intéresse à l'affranchissement de lettres et au problème de Frobenius appliqué aux timbres-poste.

1. **Affranchissement**

   Voici la tarification en vigueur pour les lettres en service rapide :

Jusqu'à	Tarif net
20 g	0,80 €
100 g	1,60 €
250 g	3,20 €
500 g	4,80 €

   On souhaite écrire une fonction `tarif(poids)` qui renvoie le tarif net du timbre en fonction du poids d'une lettre donné en grammes. Si la lettre a un poids supérieur à 500 g, la fonction renvoie `None`.

   a. Écrire la fonction en utilisant une suite d'instructions conditionnelles.

   b. Écrire la fonction en utilisant une liste de couples `[poids_max, tarif]` triée par poids croissants.
   Exemple : `[[20, 0.80], [100, 1.60], [250, 3.20], [500, 4.80]]`

2. **Augmentation**

   On suppose que l'on dispose de la liste des tarifs décrite à la question précédente. À la suite de la nouvelle année, les tarifs augmentent tous de 10 %. Modifier cette liste pour prendre en compte cette augmentation, en arrondissant à 2 chiffres après la virgule.

   Indication : `round(v, d)` arrondit la valeur `v` à `d` décimales après la virgule.

3. **Problème de Frobenius**

   On suppose que l'on ne dispose pour affranchir une lettre que de deux sortes de timbres, les uns de valeur a, les autres de valeur b (a et b strictement positifs, exprimés en centimes). On cherche à déterminer tous les tarifs réalisables à l'aide de ces deux sortes de timbres.

   Exemple : pour $a = 3$ et $b = 7$, les tarifs réalisables sont $3, 6, 7, 9, 10, 12, 13, 14, 15, \ldots$

   Le résultat est représenté par un *crible* : une liste telle que :

   - `crible[i] = 1` si la valeur i est réalisable à partir de a et b ;
   - `crible[i] = 0` sinon.

   Exemple : pour $a = 3$ et $b = 7$, on obtient `[0,0,0,1,0,0,1,1,0,1,1,0,1,1,1,1]`.

   a. On suppose que la liste `crible` est fournie. Écrire une fonction qui construit la liste des valeurs réalisables.
   Exemple : si `crible = [0,0,1,1,0,1,1,1,1,1]`, la fonction renvoie `[2,3,5,6,7,8,9]`.

   b. Pour construire la liste `crible` contenant n valeurs, on met en œuvre l'algorithme suivant (on suppose $b > a$)

   - **Étape 1 :** initialiser tous les éléments à `0`.
   - **Étape 2 :** mettre à `1` les cases d'indices a et b.
   - **Étape 3 :** parcourir les cases suivantes : si `crible[x] = 1`, alors mettre à `1` les cases d'indices $x + a$ et $x + b$ (si elles existent).

   i. Écrire une fonction qui crée la liste de taille n et l'initialise conformément aux étapes 1 et 2.

   ii. Écrire une fonction qui complète cette liste selon l'étape 3, en un seul parcours de gauche à droite.

4. **Conducteur**

   On appelle *conducteur* de a et b le plus petit entier k tel que tout entier supérieur ou égal à k soit réalisable.

   Exemple : pour $a = 3$ et $b = 7$, le conducteur est $k = 12$.

   Écrire une fonction `conducteur(crible)` qui renvoie le conducteur à partir de la liste `crible` fournie en argument. On suppose qu'il existe toujours un conducteur.

   Indication : on conseille de parcourir le tableau depuis la fin.

5. **Décomposition**

   Écrire une fonction `decompose(a, b, t)` qui, étant donné un montant `t` réalisable, renvoie le nombre de timbres de valeur a et le nombre de timbres de valeur b à utiliser, en minimisant le nombre total de timbres. On suppose $b > a$.

   Exemples : 19 est réalisable avec 1 timbre de 7 et 4 timbres de 3 ; 34 est réalisable avec 4 timbres de 7 et 2 timbres de 3.

   Indication : on teste les valeurs de i (nombre de timbres de valeur b) par ordre décroissant à partir de `t // b`. Pour chaque i, on calcule le montant restant $m = t - i \times b$; si m est divisible par a (tester avec l'opérateur `%`), on s'arrête et on a trouvé la décomposition.

Corrigé — Planche 3

1. **Affranchissement**

   a.
   ```
   def tarif(poids):
       if poids <= 20:
           return 0.8
       elif poids <= 100:
           return 1.6
       elif poids <= 250:
           return 3.2
       elif poids <= 500:
           return 4.8
   ```

   b.
   ```
   def tarif(poids):
       for p_max, tarif in [[20, 0.80], [100, 1.60], [250, 3.20], [500, 4.80]]:
           if poids <= p_max:
               return tarif
   ```

2. **Augmentation**
   ```
   for k in range(len(tarification)):
       tarification[k][1] = round(1.1*tarification[k][1],2)
   ```

3. **Problème de Frobenius**

   a.
   ```
   def valeurs_realisables(crible):
       return [k for k in range(len(cible)) if crible[k]==1]
   ```

   b.
   ```
   def genere_crible(n,a,b):
       crible = [0 for k in range(n)]
       crible[a], crible[b] = 1, 1
       for x in range(n):
           if crible[x] == 1:
               if x+a < n:
                   crible[x+a] = 1
               if x+b < n:
                   crible[x+b] = 1
       return crible
   ```

4. **Conducteur**
   ```
   def conducteur(crible):
       n = len(crible)
       for k in range(n-1,-1,-1):
           if crible[k] == 0:
               return k+1
   ```

5. **Décomposition**
   ```
   def decompose(a,b,t):
       for i in range((t//b)+1,-1,-1):
           m = t-i*b
           if m%a == 0:
               return (m//a,i)
   ```