Structures linéaires

1. Listes chaînées

On utilise ici uniquement la classe `Cellule` définie en cours :

```
class Cellule:
  '''Une cellule d'une liste chaînée'''
  def __init__(self, v, s):
    self.valeur = v
    self.suivante = s
```

On pensera à tester les fonctions définies à l'aide de `assert`.

Exercice 1
Écrire une fonction (récursive) `identique(l1, l2)` qui renvoie un booléen indiquant si les listes `l1` et `l2` sont identiques, c'est-à-dire contiennent exactement les mêmes éléments, dans le même ordre. On suppose que l'on peut comparer les éléments de `l1` et `l2` avec l'égalité == de Python.

Exercice 2
Écrire une fonction `listeN(n)` qui reçoit en argument un entier positif ou nul `n` et renvoie la liste contenant les entiers `1`, `2`, ..., `n` dans cet ordre. Si `n = 0`, la liste renvoyée est la liste vide (`None`).

Exercice 3
Écrire une fonction `affiche_liste(lst)` qui affiche, en utilisant la fonction `print`, tous les élément de la liste `lst`, séparés par des espaces. L'écrire comme une fonction récursive, puis avec une boucle `while`.

Exercice 4
Réécrire la fonction `nieme_element` avec une boucle `while`.

Exercice 5
Écrire une fonction `occurrences(x, lst)` qui renvoie le nombre d'occurrences de `x` dans `lst`. L'écrire comme une fonction récursive, puis avec une boucle `while`.

Exercice 6
Écrire une fonction `trouve(x, lst)` qui renvoie le rang de la première occurrence de `x` dans `lst`, le cas échéant, et `None` sinon. L'écrire comme une fonction récursive, puis avec une boucle `while`.

Exercice 7
Écrire une fonction `inserer(x, lst)` qui prend en arguments un entier `x` et une liste d'entiers `lst`, supposée triée par ordre croissant, et qui renvoie une nouvelle liste dans laquelle `x` a été inséré à sa place. Ainsi, insérer la valeur `3` dans la liste 1, 2, 5, 8 renvoie la liste 1, 2, 3, 5, 8.

Exercice 8
En se servant de l'exercice précédent, écrire une fonction `tri_par_insertion(lst)` qui prend en argument une liste d'entiers `lst` et renvoie une nouvelle liste, contenant les mêmes éléments et triée par ordre croissant. On suggère de l'écrire comme une fonction récursive.

Exercice 9
Écrire une fonction `liste_de_tableau(t)` qui renvoie une liste qui contient les éléments du tableau `t` (de type `list` en Python), dans le même ordre. On suggère de l'écrire avec une boucle `for`.

Exercice 10
Écrire une fonction `derniere_cellule(lst)` qui renvoie la dernière cellule de la liste `lst` et `None` si `lst` est vide.

 c'est bien une cellule qui doit être renvoyée, et pas une valeur.

Exercice 11
En utilisant la fonction de l'exercice précédent, écrire une fonction `concatener_en_place(l1,l2)` qui réalise la concaténation en place des listes `l1` et `l2`, c'est-à-dire qui relie la dernière cellule de `l1` à la première de `l2`. Cette fonction doit renvoyer la toute première cellule de la concaténation.

 Cette fonction modifie la liste `l1` (seulement) si elle n'est pas vide.

Exercice 12
Écrire une fonction `concatener(l1,l2)` qui réalise la concaténation des listes `l1` et `l2`, c'est-à-dire qui retourne une liste dont les éléments sont ceux de `l1` suivis de deux de `l2`.

 Cette fonction retourne une nouvelle liste qui contient `l2`.

Exercice 13
Écrire une fonction `renverser(lst)` qui retourne une liste dont les éléments sont ceux de `lst`, mais dans l'autre sens.

2. Piles et files

On considère la réalisation des piles avec les listes chaînées vues en cours, à reprendre pour les exercices suivants.

Exercice 1

1. Compléter la classe `Pile` avec les trois méthodes additionnelles `consulter` (qui retourne la première valeur sans l'enlever de la pile), `vider` et `taille`. Quel est l'ordre de grandeur du nombre d'opérations effectuées par la fonction `taille` ?
2. Pour éviter le problème du calcul de la taille, on se propose de revisiter la classe `Pile` en ajoutant un attribut `_taille` indiquant à tout moment la taille de la pile. Quelles méthodes doivent être modifiées et comment ?

Exercice 2 (Calculatrice Polonaise inverse à pile)

L'écriture polonaise inverse des expressions arithmétiques place l'opérateur après ses opérandes. Cette notation ne nécessite aucune parenthèse ni aucune règle de priorité. Ainsi l'expression polonaise inverse décrite par la chaîne de caractères :

```
"1 2 3 * + 4 *"
```

désigne l'expression traditionnellement notée $(1+2\times 3)\times 4$. La valeur d'une telle expression peut être calculée facilement en utilisant une pile pour stocker les résultats intermédiaires. Pour cela, on observe un à un les éléments de l'expression et on effectue les actions suivantes :

- si on voit un nombre, on le place sur la pile ;
- si on voit un opérateur binaire, on récupère les deux nombres au sommet de la pile, on leur applique l'opérateur, et on replace le résultat sur la pile.

Si l'expression était bien écrite, il y a bien toujours deux nombres sur la pile lorsque l'on voit un opérateur, et à la fin du processus il reste exactement un nombre sur la pile, qui est le résultat.

Écrire une fonction `calcP` prenant en paramètre une chaîne de caractères `expr` représentant une expression en notation polonaise inverse composée d'additions et de multiplications de nombres entiers et renvoyant la valeur de cette expression.

On supposera que les éléments de l'expression sont séparés par des espaces. On pourra donc avantageusement utiliser la méthode `expr.split(" ")` qui permet de transformer l'expression en liste de chaînes de caractères que l'on pourra parcourir.

Cette fonction ne doit pas renvoyer de résultat si l'expression est mal écrite. On laissera des erreurs se déclencher si c'est le cas.

Pour ceux qui veulent pousser plus loin, on pourra chercher comment utiliser les instructions `try ... except` et lever une exception (`raise SyntaxError("Expression mal formée")`) dans le cas où une erreur est déclenchée lors des opérations.

Exercice 3 (Parenthèse associée)

On dit qu'une chaîne de caractères comprenant, entre autres choses, des parenthèses `(` et `)` est bien parenthésée lorsque chaque parenthèse ouvrante est associé à une unique fermante, et réciproquement.

Écrire une fonction prenant en paramètres une chaîne bien parenthésée `s` et l'indice `f` d'une parenthèse fermante, et qui renvoie l'indice de la parenthèse ouvrante associée.

Indice : comme chaque parenthèse fermante est associée à la dernière parenthèse ouvrante non encore fermée, on peut suivre les associations à l'aide d'une pile.

Exercice 4 (Chaînes bien parenthésées)

On considère une chaîne de caractères incluant à la fois des parenthèses rondes `(` et `)` et des parenthèses carrées `[` et `]`. La chaîne est bien parenthésée si chaque ouvrante est associée à une unique fermante *de même forme*, et réciproquement. Écrire une fonction prenant en paramètre

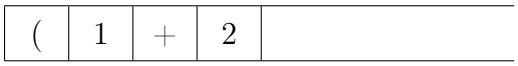
une chaîne de caractères contenant, entre autres, les parenthèses décrites et qui renvoie `True` si la chaîne est bien parenthésée et `False` sinon.

Exercice 5 (Calculatrice ordinaire)

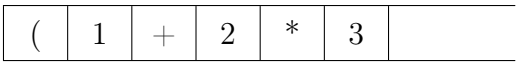
On souhaite réaliser un programme évaluant une expression arithmétique donnée par une chaîne de caractères. On utilisera les notations et les règles de priorité ordinaires, avec seulement l'addition et le produit. On supposera de plus que chaque élément est séparé des autres par un espace. Ainsi l'expression $(1 + 2 \times 3) \times 4$ sera décrite par la chaîne de caractères suivante :

`"( 1 + 2 * 3 ) * 4"`

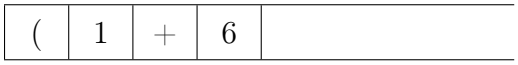
Comme dans l'exercice précédent, nous allons parcourir l'expression de gauche à droite et utiliser une pile. On alterne entre deux opérations : ajouter un nouvel élément sur la pile, et simplifier une opération présente au sommet de la pile. Ainsi dans le traitement de `"( 1 + 2 * 3 ) * 4"`, on ajoute d'abord les quatre premiers éléments pour arriver à la pile :



qui représente à son sommet l'addition $1 + 2$. Cette addition n'est pas simplifiée immédiatement car elle n'est pas prioritaire sur la multiplication qui vient ensuite. On continue à ajouter des éléments pour arriver à :



où l'on peut cette fois simplifier la multiplication `2 * 3` qui est l'opération la plus prioritaire qui soit. Le résultat est alors laissé au sommet de la pile, à la place de l'opération simplifiée :



Lorsque l'on rencontre la parenthèse fermante, l'addition `1 + 6` peut alors enfin être simplifiée à son tour, et la parenthèse ouvrante est supprimée :

On poursuit alors la progression dans l'entrée.

Pour réaliser cela, on suit un algorithme qui, pour chaque élément de l'expression en entrée, applique les critères suivants :

- Si l'élément est un nombre, on place sa valeur sur la pile.
- Si l'élément est une parenthèse `"("`, on la place sur la pile.
- Si l'élément est une parenthèse `")"`, on simplifie toutes les opérations possibles au sommet de la pile. A la fin, le sommet de la pile doit contenir un entier `n` précédé d'une parenthèse ouvrante `"("`, parenthèse que l'on retire pour ne garder que `n`.
- Si l'élément est un opérateur (`"+"` ou `"*"`), on simplifie toutes les opérations au sommet de la pile utilisant des opérateurs aussi prioritaires ou plus prioritaires que le nouvel opérateur, puis on place ce dernier sur la pile.

Une fois l'expression totalement lue, on simplifie les opérations de la pile pour n'obtenir qu'un nombre.

Écrire une fonction `simplifie(pile, ops)` qui, tant qu'il y a des opérations de la liste `ops` au sommet de la pile (après la valeur du sommet), effectue ces opérations pour réduire la pile.

Écrire ensuite une fonction `calcule(expression)` renvoyant la valeur de l'expression représentée par la chaîne de caractères `expression` donnée en paramètre.

Il est à noter que l'utilisation de piles non homogènes (ici on mélange des nombres avec des opérations et des parenthèses) n'est généralement pas conseillé.

Exercice 6 (Pile bornée)
Une pile bornée est une pile dotée à sa création d'une capacité maximale.
On propose l'interface suivante :

fonction	description
`creer_pile(c)`	crée et renvoie une pile bornée de capacité `c`
`est_vide(p)`	renvoie `True` si la pile est vide et `False` sinon
`est_pleine(p)`	renvoie `True` si la pile est pleine et `False` sinon
`empiler(p, e)`	ajoute `e` au sommet de `p` si `p` n'est pas pleine, et lève une exception `IndexError` sinon
`depiler(p)`	retire et renvoie l'élément au sommet de `p` si `p` n'est pas vide, et lève une exception `Indexerror` sinon

Réaliser cette interface après avoir défini une classe `PileBornee`.

Exercice 7 (File bornée)
Une file bornée est une file dotée à sa création d'une capacité maximale.
On propose l'interface suivante :

fonction	description
`creer_file(c)`	crée et renvoie une file bornée de capacité `c`
`est_vide(f)`	renvoie `True` si la file est vide et `False` sinon
`est_pleine(f)`	renvoie `True` si la file est pleine et `False` sinon
`ajouter(f, e)`	ajoute `e` àl'arrière de `f` si `f` n'est pas pleine, et lève une exception `IndexError` sinon
`retirer(f)`	retire et renvoie l'élément à l'avant de `f` si `f` n'est pas vide, et lève une exception `Indexerror` sinon

Réaliser cette interface après avoir défini une classe `FileBornee`.

Exercice 8
Dans cet exercice, on se propose d'évaluer le temps d'attente de clients à des guichets, en comparant la solution d'une unique file d'attente et la solution d'une file d'attente par guichet. Pour cela, on modélise le temps par une variable globale, qui est incrémentée à chaque tour de boucle.
Lorsqu'un nouveau client arrive, il est placé dans une file sous la forme d'un entier égal à la valeur de l'horloge, c'est-à-dire égale à son heure d'arrivée.
Lorsqu'un client est servi, c'est-à-dire lorsqu'il sort de sa file d'attente, on obtient son temps d'attente en faisant une soustraction de la valeur courante de l'horloge et de la valeur qui vient d'être retirée de la file.
L'idée est de faire tourner une telle simulation relativement longtemps, tout en totalisant le nombre de clients servis et le temps d'attente cumulé sur tous les clients. Le rapport de ces deux quantités nous donne le temps d'attente moyen. On peut alors comparer plusieurs stratégies (une ou plusieurs files, choix d'une file au hasard quand il y en a plusieurs, choix de la file ou il y a le moins de clients, etc.).
On se donne un nombre `N` de guichets (par exemple, `N=5`). pour simuler la disponibilité d'un guichet, on peut se donner un tableau d'entiers `dispo` de taille `N`. La valeur `dispo[i]` indique le nombre de tours d'horloge où le guichet `i` sera occupé. en particulier, lorsque cette valeur vaut 0, cela veut dire que le guichet est libre et peut donc servir un nouveau client.
Lorsqu'un client est servi par le guichet `i`, on choisit un temps de traitement pour ce client, au hasard entre `0` et un autre entier `M` (par exemple `M=5`), et on l'affecte à `dispo[i]`. A chaque tour d'horloge, on réalise deux opérations :

- on fait apparaître un nouveau client
- pour chaque guichet `i`,

* s'il est disponible, il sert un nouveau client (pris dans sa propre file ou dans l'unique file, selon le modèle), le cas échéant ;
* sinon, on décrémente `dispo[i]`.

Écrire un programme qui effectue une telle simulation, sur 100 000 tours d'horloge, et affiche au final le temps d'attente moyen. Comparer avec différentes stratégies.

Exercice 9 (Épreuve pratique)

Cet exercice utilise des piles qui seront représentées en Python par des listes (type `list`). On rappelle que l'expression `T1 = list(T)` fait une copie de `T` indépendante de `T`, que l'expression `x = T.pop()` enlève le sommet de la pile `T` et le place dans la variable `x` et, enfin, que l'expression `T.append(v)` place la valeur `v` au sommet de la pile `T`.

Compléter le code Python de la fonction `positif` ci-dessous qui prend une pile `T` de nombres entiers en paramètre et qui renvoie la pile des entiers positifs dans le même ordre, sans modifier la variable `T`.

```
def positif(T):
  T2 = ...(T)
  T3 = ...
  while T2 != []:
    x = ...
    if ... >= 0:
      T3.append(...)
  T2 = []
  while T3 != ...:
    x = T3.pop()
    ...
  print("T =",T)
  return T2
```

Exemple :

```
>>> positif([-1,0,5,-3,4,-6,10,9,-8])
T = [-1, 0, 5, -3, 4, -6, 10, 9, -8]
[0, 5, 4, 10, 9]
```

Exercice 10 (Épreuve pratique)

On veut écrire une classe pour gérer une file à l'aide d'une liste chaînée. On dispose d'une classe Maillon permettant la création d'un maillon de la chaîne, celui-ci étant constitué d'une valeur et d'une référence au maillon suivant de la chaîne :

```
class Maillon:
  def __init__(self,v):
    self.valeur = v
    self.suivant = None
```

Compléter la classe File suivante où l'attribut `dernier_file` contient le maillon correspondant à l'élément arrivé en dernier dans la file :

```
class File:
  def __init__(self):
    self.dernier_file = None

  def enfile(self,element):
    nouveau_maillon = Maillon(...)
    nouveau_maillon.suivant = self.dernier_file
    self.dernier_file = ...

  def est_vide(self):
    return self.dernier_file == None

  def affiche(self):
    maillon = self.dernier_file
    while maillon != ... :
      print(maillon.valeur)
      maillon = ...

  def defile(self):
    if not self.est_vide() :
      if self.dernier_file.suivant == None :
        resultat = self.dernier_file.valeur
        self.dernier_file = None
        return resultat
      maillon = ...
      while maillon.suivant.suivant != None :
        maillon = maillon.suivant
      resultat = ...
      maillon.suivant = None
      return resultat
    return None
```

On pourra tester le fonctionnement de la classe en utilisant les commandes suivantes dans la console Python :

```
>>> F = File()
>>> F.est_vide()
True
>>> F.enfile(2)
>>> F.affiche()
2
>>> F.est_vide()
False
>>> F.enfile(5)
>>> F.enfile(7)
```

```
>>> F.affiche()
7
5
2
>>> F.defile()
2
>>> F.defile()
5
>>> F.affiche()
7
```

3. Dictionnaires

Exercice 1
On considère le programme ci-dessous :

```
tab = []
duree_trajet = {"pied":55, "velo":20, "tram":30, "voiture":25}
for mt,tps in duree_trajet.items():
  if tps < 30:
    tab.append(mt)
```

Quelle est le contenu de `tab` après l'exécution de ce programme ?

Exercice 2
On utilise un dictionnaire pour stocker les notes des élèves où les clefs sont les noms des élèves et les valeurs associés sont les notes :

```
notes = {
  "Balthazar":18,
  "Melchior":20,
  "Gaspard":5
  }
```

1. Compléter la fonction `moyenne(notes)` qui prend en paramètre un dictionnaire `notes` et retourne la moyenne des notes.

```
def moyenne(notes):
  s = 0
  for note in ...:
    s += note
  return ...
```

2. Compléter la fonction `meilleurs_eleves(notes)` qui prend en paramètre un dictionnaire `notes` et retourne les noms des élèves qui ont la meilleur note.

```
def meilleurs_eleves(notes):
  noms = []
  max_note = 0
  for nom, note in ... :
    if ... > max_note:
      noms = [...]
      ... = ...
    elif ... == max_note:
      noms.append(...)
  return noms
```

Exercice 3
On utilise un dictionnaire pour stocker les bulletins des élèves qui associe à chaque nom un dictionnaire contenant les notes obtenues par matière :

```
bulletins = {
  "Balthazar":{"NSI":18, "Mathématiques":16, "Philosophie":9},
  "Melchior":{"NSI":20, "Mathématiques":18, "Philosophie":15},
  "Gaspard":{"NSI":5, "Mathématiques":10, "Philosophie":14},
  }
```

1. Compléter la fonction `moyennes(bulletins)` qui prend en paramètre un dictionnaire `bulletin` et retourne un dictionnaire des moyennes où les clefs sont les noms des élèves et les valeurs leur moyenne.
   On pourra utiliser la fonction moyenne de l'exercice précédent.

```
def moyennes(bulletins):
  m = {}
  for nom, bulletin in bulletins.items():
    m[...] = ...
  return m
```

2. Compléter la fonction `bulletin_matiere` qui prend en paramètres un dictionnaire `bulletins` et une chaîne de caractère `nom_matiere` et retourne un dictionnaire dont les clefs sont les noms des élèves et les valeurs les notes obtenues à la matière nommée.
   Exemple avec le dictionnaire précédent :

```
>>> bulletin_matiere(bulletins, "NSI")
{"Balthazar":18, "Melchior":20, "Gaspard":5}
```

```
def bulletin_matiere(bulletins, nom_matiere):
  bm = {}
  for nom in ...:
    bm[...] = bulletins[...][...]
  return bm
```

Exercice 4

On considère un dictionnaire `personnes` qui associe à des noms de personnes un dictionnaire contenant des informations personnelles :

```
personnes = {
  "Jean Aymar":{"taille":178, "pays":"USA", "age":31},
  "Clio Patre":{"pays":"Portugal", "age":32, "taille":179}
}
```

1. Écrire une fonction `age(personnes, nom)` qui prend un dictionnaire `personnes` et un nom de personne en paramètres et retourne son age si la personne est dans le dictionnaire et `None` sinon.
2. Écrire une fonction `taille_moyenne(personnes)` qui retourne la taille moyenne des personnes dans le dictionnaires `personnes`.

Exercice 5

Écrire une fonction `nb_occurrences` qui crée un dictionnaire à partir d'une chaîne de caractères. Les clefs sont les lettres de cette chaîne de caractères et les valeurs les occurrences de ces lettres.
Exemple :

```
>>> nb_occurrences("banane")
{'b':1, 'a':2, 'n':2, 'e':1}
```

Exercice 6
Écrire une fonction `ajouter` qui prend en paramètres deux dictionnaires et retourne un dictionnaire où les valeurs sont obtenues en ajoutant les valeurs des clés communes.
Exemple :

```
>>> d1 = {"a":100, "b":200, "c":300}
>>> d2 = {"a":300, "b":200, "d":400}
>>> ajouter(d1, d2)
{"a":400, "b":400, "d":400, "c":300}
```

Exercice 7
Écrire une fonction `combine` qui combine deux dictionnaires en créant une liste de valeurs pour chaque clé.
Exemple :

```
>>> d1 = {"w":50, "x":100, "y":"Vert", "z":400}
>>> d2 = {"x":300, "y":"Rouge", "z":600}
>>> combiner(d1, d2)
{"w":[50], "x":[100, 300], "y":["Vert", "Rouge"], "z":[400, 600]}
```

4. Annales

EXERCICE 4 (4 points)

Cet exercice traite du thème « structures de données », et principalement des piles.

La classe `Pile` utilisée dans cet exercice est implémentée en utilisant des listes Python et propose quatre éléments d'interface :

- Un constructeur qui permet de créer une pile vide, représentée par `[]` ;
- La méthode `est_vide()` qui renvoie `True` si l'objet est une pile ne contenant aucun élément, et `False` sinon ;
- La méthode `empiler` qui prend un objet quelconque en paramètre et ajoute cet objet au sommet de la pile. Dans la représentation de la pile dans la console, cet objet apparaît à droite des autres éléments de la pile ;
- La méthode `depiler` qui renvoie l'objet présent au sommet de la pile et le retire de la pile.

Exemples :

```
>>> mapile = Pile()
>>> mapile.empiler(2)
>>> mapile
[2]
>>> mapile.empiler(3)
>>> mapile.empiler(50)
>>> mapile
[2, 3, 50]
>>> mapile.depiler()
50
>>> mapile
[2, 3]
```

La méthode `est_triee` ci-dessous renvoie `True` si, en dépilant tous les éléments, ils sont traités dans l'ordre croissant, et `False` sinon.

```
1 def est_triee(self):
2   if not self.est_vide() :
3      e1 = self.depiler()
4      while not self.est_vide():
5          e2 = self.depiler()
6          if e1 ... e2 :
7              return False
8          e1 = ...
9     return True
```

1. Recopier sur la copie les lignes 6 et 8 en complétant les points de suspension.

On créé dans la console la pile `A` représentée par `[1, 2, 3, 4]`.

2. a. Donner la valeur renvoyée par l'appel `A.est_triee()`.
   b. Donner le contenu de la pile `A` après l'exécution de cette instruction.

On souhaite maintenant écrire le code d'une méthode `depileMax` d'une pile non vide ne contenant que des nombres entiers et renvoyant le plus grand élément de cette pile en le retirant de la pile.
Après l'exécution de `p.depileMax()`, le nombre d'éléments de la pile `p` diminue donc de 1.

```
1   def depileMax(self):
2       assert not self.est_vide(), "Pile vide"
3       q = Pile()
4       maxi = self.depiler()
5       while not self.est_vide() :
6          elt = self.depiler()
7          if maxi < elt :
8             q.empiler(maxi)
9             maxi = ...
10          else :
11             ...
12       while not q.est_vide():
13          self.empiler(q.depiler())
14       return maxi
```

3. Recopier sur la copie les lignes 9 et 11 en complétant les points de suspension.

On créé la pile `B` représentée par [9, -7, 8, 12, 4] et on effectue l'appel `B.depileMax()`.

4. a. Donner le contenu des piles `B` et `q` à la fin de chaque itération de la boucle `while` de la ligne 5.
   b. Donner le contenu des piles `B` et `q` avant l'exécution de la ligne 14.
   c. Donner un exemple de pile qui montre que l'ordre des éléments restants n'est pas préservé après l'exécution de `depileMax`.

On donne le code de la méthode `traiter()` :

```
1   def traiter(self):
2       q = Pile()
3       while not self.est_vide():
4           q.empiler(self.depileMax())
5       while not q.est_vide():
6           self.empiler(q.depiler())
```

5. a. Donner les contenus successifs des piles `B` et `q`
   - avant la ligne 3,
   - avant la ligne 5,
   - à la fin de l'exécution de la fonction `traiter`

   lorsque la fonction `traiter` est appliquée sur la pile `B` contenant [1, 6, 4, 3, 7, 2].

   b. Expliquer le traitement effectué par cette méthode.

EXERCICE 1 (4 points)

Cet exercice composé de deux parties A et B, porte sur les structures de données.

Partie A : Expression correctement parenthésée

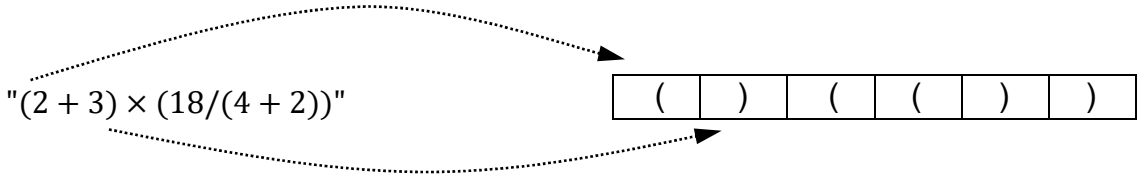
On veut déterminer si une expression arithmétique est correctement parenthésée.

Pour chaque parenthèse fermante ")" correspond une parenthèse précédemment ouverte "(".

Exemples :

- L'expression arithmétique "$(2+3)\times(18/(4+2))$" est correctement parenthésée.
- L'expression arithmétique "$(2+3)\times(18/(4+2$" est non correctement parenthésée.

Pour simplifier les expressions arithmétiques, on enregistre, dans une structure de données, uniquement les parenthèses dans leur ordre d'apparition. On appelle expression simplifiée cette structure.



1. Indiquer si la phrase « les éléments sont maintenant retirés (pour être lus) de cette structure de données dans le même ordre qu'ils y ont été ajoutés lors de l'enregistrement » décrit le comportement d'une file ou d'une pile. Justifier.

Pour vérifier le parenthésage, on peut utiliser une variable `controleur` qui :

- est un nombre entier égal à 0 en début d'analyse de l'expression simplifiée ;
- augmente de 1 si l'on rencontre une parenthèse ouvrante "(" ;
- diminue de 1 si l'on rencontre une parenthèse fermante ")".

Exemple : On considère l' expression simplifiée A : "( )(( ))"
Lors de l'analyse de l'expression A, `controleur` (initialement égal à 0) prend successivement pour valeur 1, 0, 1, 2, 1, 0. Le parenthésage est correct.

2. Écrire, pour chacune des 2 expressions simplifiées B et C suivantes, les valeurs successives prises par la variable `controleur` lors de leur analyse.
Expression simplifiée B : " (((( )( )"
Expression simplifiée C : "(( ))("

3. L'expression simplifiée B précédente est mal parenthésée (parenthèses fermantes manquantes) car le `controleur` est différent de zéro en fin d'analyse. L'expression simplifiée C précédente est également mal parenthésée (parenthèse fermante sans parenthèse ouvrante) car le `controleur` prend une valeur négative pendant l'analyse.
Recopier et compléter uniquement les lignes 13 et 16 du code ci-dessous pour que la fonction `parenthesage_correct` réponde à sa description.

```
1 def parenthesage_correct(expression):
2    ''' fonction retournant True si l'expression arithmétique
3    simplifiée (str) est correctement parenthésée, False
4    sinon.
5    Condition: expression ne contient que des parenthèses
6    ouvrantes et fermantes '''

7   controleur = 0
8   for parenthese in expression: #pour chaque parenthèse
9      if parenthese == '(':
10        controleur = controleur + 1
11     else:# parenthese == ')'
12        controleur = controleur - 1
13        if controleur ...  : # test 1 (à recopier et compléter)
14           #parenthèse fermante sans parenthèse ouvrante
15           return False
16   if controleur ... : # test 2  (à recopier et compléter)
17      return True #le parenthésage est correct
18   else:
19      return False #parenthèse(s) fermante(s) manquante(s)
```

Partie B : Texte correctement balisé

On peut faire l'analogie entre le texte simplifié des fichiers HTML (uniquement constitué de balises ouvrantes <nom> et fermantes </nom>) et les expressions parenthésées :

Par exemple, l'expression HTML simplifiée :
" " est correctement balisée.

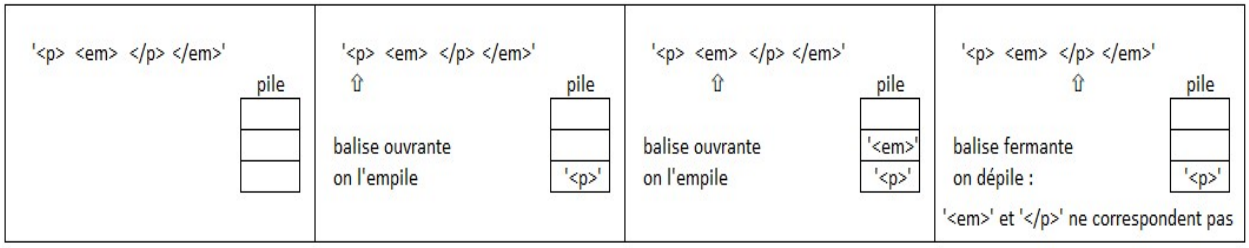
On ne tiendra pas compte dans cette partie des balises ne comportant pas de fermeture comme
 ou <img …>.

Afin de vérifier qu'une expression HTML simplifiée est correctement balisée, on peut utiliser une pile (initialement vide) selon l'algorithme suivant :

On parcourt successivement chaque balise de l'expression :

- lorsque l'on rencontre une balise ouvrante, on l'empile ;
- lorsque l'on rencontre une balise fermante :
  - si la pile est vide, alors l'analyse s'arrête : le balisage est incorrect ,
  - sinon, on dépile et on vérifie que les deux balises (la balise fermante rencontrée et la balise ouvrante dépilée) correspondent (c'est-à-dire ont le même nom) si ce n'est pas le cas, l'analyse s'arrête (balisage incorrect).

Exemple : État de la pile lors du déroulement de cet algorithme pour l'expression simplifiée `"<p><em></p></em>"` qui n'est pas correctement balisée.



État de la pile lors du déroulement de l'algorithme

4. Cette question traite de l'état de la pile lors du déroulement de l'algorithme.

a. Représenter la pile à chaque étape du déroulement de cet algorithme pour l'expression `"<p><em></em></p>"` (balisage correct).

b. Indiquer quelle condition simple (sur le contenu de la pile) permet alors de dire que le balisage est correct lorsque toute l'expression HTML simplifiée a été entièrement parcourue, sans que l'analyse ne s'arrête.

5. Une expression HTML correctement balisée contient 12 balises.
Indiquer le nombre d'éléments que pourrait contenir au maximum la pile lors de son analyse.

EXERCICE 2 (4 points)

Cet exercice porte sur les structures de données.

La poussette est un jeu de cartes en solitaire. Cet exercice propose une version simplifiée de ce jeu basée sur des nombres.

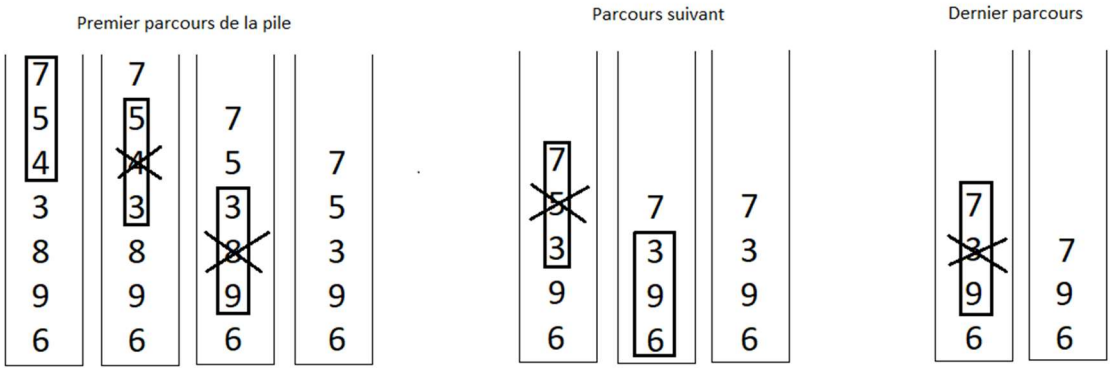
On considère une pile constituée de nombres entiers tirés aléatoirement. Le jeu consiste à réduire la pile suivant la règle suivante : quand la pile contient du haut vers le bas un triplet dont les termes du haut et du bas sont de même parité, on supprime l'élément central.

Par exemple :

- Si la pile contient du haut vers le bas, le triplet 1 0 3, on supprime le 0.
- Si la pile contient du haut vers le bas, le triplet 1 0 8, la pile reste inchangée.

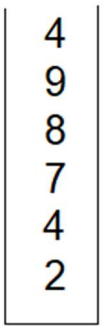
On parcourt la pile ainsi de haut en bas et on procède aux réductions. Arrivé en bas de la pile, on recommence la réduction en repartant du sommet de la pile jusqu'à ce que la pile ne soit plus réductible. Une partie est « gagnante » lorsque la pile finale est réduite à deux éléments exactement.

Voici un exemple détaillé de déroulement d'une partie.

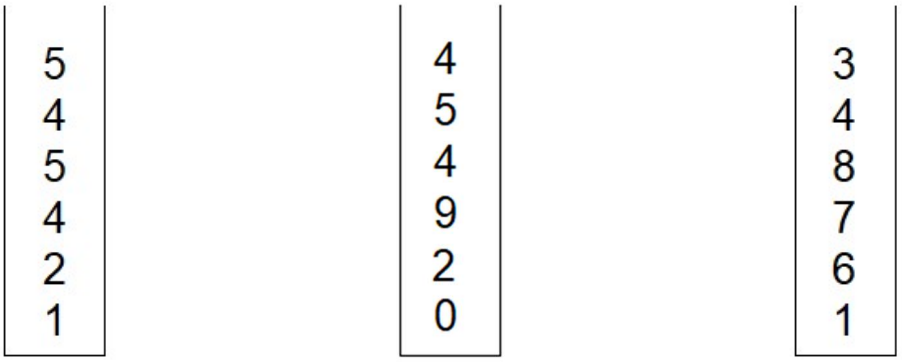


1.

a. Donner les différentes étapes de réduction de la pile suivante :



b. Parmi les piles proposées ci-dessous, donner celle qui est gagnante.



Pile A Pile B Pile C

L'interface d'une pile est proposée ci-dessous. On utilisera uniquement les fonctions figurant dans le tableau suivant :

Structure de données abstraite : Pile
• `creer_pile_vide()` renvoie une pile vide
• `est_vide(p)` renvoie `True` si `p` est vide, `False` sinon
• `empiler(p, element)` ajoute `element` au sommet de `p`
• `depiler(p)` retire l'élément au sommet de `p` et le renvoie
• `sommet(p)` renvoie l'élément au sommet de `p` sans le retirer de `p`
• `taille(p)` : renvoie le nombre d'éléments de `p`

2. La fonction `reduire_triplet_au_sommet` permet de supprimer l'élément central des trois premiers éléments en partant du haut de la pile, si l'élément du bas et du haut sont de même parité. Les éléments dépilés et non supprimés sont replacés dans le bon ordre dans la pile.
Recopier et compléter sur la copie le code de la fonction `reduire_triplet_au_sommet` prenant une pile `p` en paramètre et qui la modifie en place. Cette fonction ne renvoie donc rien.

```
1  def reduire_triplet_au_sommet(p):
2      a = depiler(p)
3      b = depiler(p)
4      c = sommet(p)
5      if a % 2 != .... :
6          empiler(p, ...)
7      empiler(p, ...)
```

3. On se propose maintenant d'écrire une fonction `parcourir_pile_en_reduisant` qui parcourt la pile du haut vers le bas en procédant aux réductions pour chaque triplet rencontré quand cela est possible.

a. Donner la taille minimale que doit avoir une pile pour être réductible.

b. Recopier et compléter sur la copie :

```
1  def parcourir_pile_en_reduisant(p):
2      q = creer_pile_vide()
3      while taille(p) >= ....:
4          reduire_triplet_au_sommet(p)
5          e = depiler(p)
6          empiler(q, e)
7      while not est_vide(q):
8          ............
9          ............
10      return p
```

4. Partant d'une pile d'entiers `p`, on propose ici d'implémenter une fonction récursive `jouer` renvoyant la pile `p` entièrement simplifiée. Une fois la pile parcourue de haut en bas et réduite, on procède à nouveau à sa réduction à condition que cela soit possible. Ainsi :

- Si la pile `p` n'a pas subi de réduction, on la renvoie.
- Sinon on appelle à nouveau la fonction `jouer` , prenant en paramètre la pile réduite.

Recopier et compléter sur la copie le code ci-dessous :

```
1  def jouer(p):
2      q = parcourir_pile_en_reduisant(p)
3      if ............. :
4          return p
5      else:
6          return jouer(...)
```

EXERCICE 2 (4 points)

Cet exercice porte sur les structures de données (files et la programmation objet en langage python)

Un supermarché met en place un système de passage automatique en caisse.
Un client scanne les articles à l'aide d'un scanner de code-barres au fur et à mesure qu'il les ajoute dans son panier.
Les articles s'enregistrent alors dans une structure de données.

La structure de données utilisée est une file définie par la classe `Panier`, avec les primitives habituelles sur la structure de file.
Pour faciliter la lecture, le code de la classe `Panier` n'est pas écrit.

```
class Panier():
  def __init__(self):
     """Initialise la file comme une file vide."""

  def est_vide(self):
     """Renvoie True si la file est vide, False sinon."""

  def enfiler(self, e):
     """Ajoute l'élément e en dernière position de la file,
    ne renvoie rien."""

  def defiler(self):
     """Retire le premier élément de la file et le renvoie."""
```

Le panier d'un client sera représenté par une file contenant les articles scannés.
Les articles sont représentés par des tuples `(code_barre, designation, prix, horaire_scan)` où

- `code_barre` est un nombre entier identifiant l'article ;
- `designation` est une chaine de caractères qui pourra être affichée sur le ticket de caisse ;
- `prix` est un nombre décimal donnant le prix d'une unité de cet article ;
- `horaire_scan` est un nombre entier de secondes permettant de connaitre l'heure où l'article a été scanné.

1. On souhaite ajouter un article dont le tuple est le suivant `(31002, "café noir", 1.50, 50525)`.
   Ecrire le code utilisant une des quatre méthodes ci-dessus permettant d'ajouter l'article à l'objet de classe `Panier` appelé `panier1`.

2. On souhaite définir une **méthode** `remplir(panier_temp)` dans la classe `Panier` permettant de remplir la file avec tout le contenu d'un autre panier `panier_temp` qui est un objet de type `Panier`.

Recopier et compléter le code de la méthode `remplir` en remplaçant chaque `.....` par la primitive de file qui convient.

```
def remplir(self, panier_temp):
    while not panier_temp.....  :
        article = panier_temp.....
        self.....(article)
```

3. Pour que le client puisse connaitre à tout moment le montant de son panier, on souhaite ajouter une **méthode** `prix_total()` à la classe `Panier` qui renvoie la somme des prix de tous les articles présents dans le panier.
Ecrire le code de la méthode `prix_total`. **Attention, après l'appel de cette méthode, le panier devra toujours contenir ses articles**.

4. Le magasin souhaite connaitre pour chaque client la durée des achats. Cette durée sera obtenue en faisant la différence entre le champ `horaire_scan` du dernier article scanné et le champ `horaire_scan` du premier article scanné dans le panier du client. Un panier vide renverra une durée égale à zéro. On pourra accepter que le panier soit vide après l'appel de cette méthode.
Ecrire une **méthode** `duree_courses` de la classe `Panier` qui renvoie cette durée.