1) *Utilisation d'une pile pour tester le parenthésage d'une expression*

Comme l'indique l'énoncé, je parcours de gauche à droite la chaîne `ep` représentant l'expression.

- lorsque je rencontre une parenthèse ouvrante, j'empile sa position
- lorsque je rencontre une parenthèse fermante, la position de la parenthèse ouvrante correspondante est censée être au sommet de la pile, donc si la pile est vide je renvoie un message d'erreur, sinon je dépile et j'affiche le couple correct que je viens de détecter (option suggérée par l'énoncé)
- à la fin du parcours, la pile doit être vide (toutes les parenthèses ouvrantes ont bien été refermées) : si c'est le cas, je valide le parenthésage, sinon je retourne un message d'erreur.

Voici donc le programme, gestion des erreurs incluse !

```
def tester(ep):
    p = pile_vide()
    for j in range(len(ep)):
        if ep[j] == '(':
            empiler(j,p)
        elif ep[j] == ')':
            if est_vide(p):
                return ") à l'indice " + str(j) + " n'est pas ouverte !"
            else:
                i = depiler(p)
                print("Couple ", i, " ", j, " correct")
    if est_vide(p):
        return "Parenthésage OK"
    else:
        return "( à l'indice " + str(sommet(p)) + " n'est pas refermée !"
```

2) *Implémentation d'une file d'attente*

a) Le problème de la simple liste est que seuls les ajouts et suppressions en fin de liste s'effectuent en temps constant. Or pour une file d'attente, l'ajout d'un nouvel élément doit se faire à un bout de la liste et la suppression à l'autre bout. Donc l'une des deux opérations pourra s'exécuter en temps constant (amorti), mais l'autre se fera en temps linéaire par rapport à la longueur de la liste utilisée.

Je choisis par exemple d'ajouter chaque nouvel élément en fin de liste et de récupérer le "sortant" en début de liste. L'affichage consiste alors simplement à imprimer la liste (le prochain sortant étant la tête de la liste). Cela donne les primitives suivantes :

```
def file_vide():
    return []

def est_file_vide(f):
    return f == []

def suivant(f):
    s = f[0]
    for k in range(1, len(f)):
        f[k-1] = f[k]
    f.pop()
    return s
```

```
def ajouter(c, f):
    f.append(c)
    return f

def afficher(f):
    print(f)
```

Noter que la fonction `suivant` pourrait se résumer à `return f.pop(0)`, mais la méthode `pop` n'est au programme que dans sa version sans paramètre, qui correspond à la suppression en temps constant **du dernier élément**. Cela pour éviter d'occulter les complexités "cachées" ; par exemple la complexité de `pop(0)` est linéaire en la longueur de la liste.

b) Selon l'indication de l'énoncé, je définis une file `f` comme étant une liste de deux piles :

∗ la première (`f[0]`) contient le début de la file, le premier client à servir se trouvant au sommet ;
∗ la deuxième (`f[1]`) contient la fin de la file, le dernier client entré se trouvant au sommet.

Pour respecter le fonctionnement LIFO, l'ajout d'un nouveau client s'effectue en l'empilant sur `f[1]`, tandis que l'extraction du premier client à servir s'effectue en le dépilant de `f[0]`, cela bien sûr à condition que `f[0]` ne soit pas vide ! Si elle l'est, je commence par "transvaser" les éléments de `f[1]` dans `f[0]`, en sens inverse pour que le dernier arrivé soit bien le dernier servi.

Pour l'affichage, `print(f)` montrera les deux listes, pas évidentes à interpréter. C'est pourquoi une fonction d'affichage pourra être utile : je conviens d'afficher une unique liste, commençant par le premier client à servir et se terminant par le dernier arrivé. Compte tenu du mode de stockage d'une pile (une liste avec le sommet à la fin !), je construis la liste à afficher en retournant `f[0]` (méthode `reverse()`) et en ajoutant `f[1]` à la fin de la liste obtenue (méthode `extend()`). Ces méthodes sont *a priori* hors programme, mais je m'autorise à les utiliser ici puisqu'il s'agit de programmer un outil qui ne fait pas partie des primitives officielles du type "file". Bien sûr on peut programmer cela à l'aide de boucles.

Quant à la complexité, `ajouter` s'exécute en temps constant amorti (cf. la méthode `append()` des listes Python) et `suivant` s'exécute aussi en temps constant amorti, au sens où elle s'exécute "en général" en temps constant (cf. la méthode `pop()`), avec "exceptionnellement" le transvasement en temps linéaire par rapport à la longueur de la file.

```
def pile_vide():
    return []

def est_pile_vide(p):
    return p == []

def sommet(p):
    return p[-1]

def empiler(s, p):
    p.append(s)
    return p

def depiler(p):
    return p.pop()

def file_vide():
    return [[], []]
```

```
def est_file_vide(f):
    return est_pile_vide(f[0]) \
           and est_pile_vide(f[1])

def suivant(f):
    if est_pile_vide(f[0]):
        while not est_pile_vide(f[1]):
            empiler(depiler(f[1]), f[0])
    return depiler(f[0])

def ajouter(c, f):
    empiler(c, f[1])
    return f

def afficher(f):
    file=f[0].copy()
    file.reverse()
    file.extend(f[1])
    print(file)
```

3) Comme l'énoncé le propose, je me limite aux expressions contenant des additions et multiplications d'entiers. Les programmes s'adaptent facilement aux autres cas.

Le programme page suivante suppose définies les primitives du type pile.

Chaque symbole est une chaîne de caractères, représentant soit un signe opératoire, soit un nombre.

Pour respecter la définition du type pile, je m'interdis de calculer la "longueur" d'une pile (alors que ce serait facile compte tenu de l'implémentation choisie !). La seule mesure de taille pour une pile est le test `est_pile_vide` ! Ce test est suffisant pour s'assurer que la pile n'est pas vide avant de dépiler...

En revanche, pour les symboles autres que les signes opératoires, le test permettant de savoir si la chaîne représente bien un nombre est plus compliqué (une boucle suffirait pour un entier, mais dans le cas d'un flottant il faut prendre en compte les différents formats...), d'où l'intérêt de l'instruction `try` (hors programme !), qui permet d'intercepter une erreur éventuelle. Son utilisation est assez simple, à condition de connaître le nom de l'erreur risquant d'être déclenchée (ici `ValueError`), afin d'afficher le message correspondant. Mais aux concours, sauf indication contraire, il faudra imposer comme précondition que les symboles fournis soient corrects !

```
def eval_avec_controle(eap):
    symbols = eap.split()
    p = pile_vide()
    for s in symbols:
        if s == '+':
            if est_vide(p):
                return 'Pas assez de nombres'
            a = depiler(p)
            if est_vide(p):
                return 'Pas assez de nombres'
            b = depiler(p)
            empiler(int(a)+int(b), p)
        elif s == '*':
            if est_vide(p):
                return 'Pas assez de nombres'
            a = depiler(p)
            if est_vide(p):
                return 'Pas assez de nombres'
            b = depiler(p)
            empiler(int(a)*int(b), p)
        else:
            try:
                empiler(int(s), p)
            except ValueError:
                return 'Erreur de saisie'
    a = depiler(p)
    if est_vide(p):
        return a
    else:
        return 'Trop de nombres'
```

4) **a)** Il y a quatre cases à tester, sans oublier de vérifier qu'elles sont bien vides (valeur 1) !

```
def casesVoisines(laby, i, j):
    n = len(laby)
    lst = []
    if i > 0 and laby[i-1][j] == 1:
        lst.append((i-1, j))
    if i < n-1 and laby[i+1][j] == 1:
        lst.append((i+1, j))
    if j > 0 and laby[i][j-1] == 1:
        lst.append((i, j-1))
    if j < n-1 and laby[i][j+1] == 1:
        lst.append((i, j+1))
    return lst
```

b) Je traduis l'algorithme de l'énoncé.

Le résultat booléen souhaité est obtenu en testant la valeur de la case de sortie.

Comme chaque case est empilée au plus une fois, la complexité est un $O\left(n^2\right)$.

On pourrait renvoyer `True` (et donc stopper l'exécution) dès que l'on marque la case de sortie, mais l'énoncé demande expressément de marquer toutes les cases accessibles depuis l'entrée, cela pour les applications futures... De toute façon cela ne modifierait pas l'ordre de grandeur de la complexité en moyenne.

```
def parcours(laby):
    n = len(laby)
    p = pile_vide()
    empiler((0, 1), p)
    while not est_pile_vide(p):
        x, y = depiler(p)
        casesAVisiter = casesVoisines(laby, x, y)
        for (i, j) in casesAVisiter :
            if laby[i][j] < 2 :
                laby[i][j] = 2
                empiler ((i, j), p)
    return laby[n-1][n-2] == 2
```

5) a) Je commence par créer une liste de listes pleines de 0, puis je place aléatoirement des 1 (cases vides) dans les lignes de 2 à $n-3$, sans toucher aux bords. Enfin je remplis les lignes 1 et $n-2$ avec des 1 (sauf les bords) et je place la case d'entrée et la case de sortie du labyrinthe.

```
def creerMilieu(n, p):
    milieuPoreux = [[0] * n for i in range(n)]
    for i in range(2, n - 2):
        for j in range(1, n - 1):
            if rd.random() < p:
                milieuPoreux[i][j] = 1
    # Seconde et avant-dernière ligne :
    for j in range(1, n - 1):
        milieuPoreux[1][j] = 1
        milieuPoreux[n - 2][j] = 1
    # Entrée et sortie du milieu :
    milieuPoreux[0][1] = 1
    milieuPoreux[n - 1][n - 2] = 1
    return milieuPoreux
```

b) J'utilise l'indication présente à la fin de la question **4.** Ici j'utilise la fonction `parcours` comme une procédure, le but étant d'afficher les cases accessibles et non de savoir si l'eau a réussi à traverser le milieu poreux.

```
def afficherEcoulement(n, p):
    milieuPoreux = creerMilieu(n, p)
    parcours(milieuPoreux)
    palette = colors.ListedColormap(['black', 'white','blue'])
    plt.matshow(milieuPoreux,cmap = palette)
    plt.show()
```

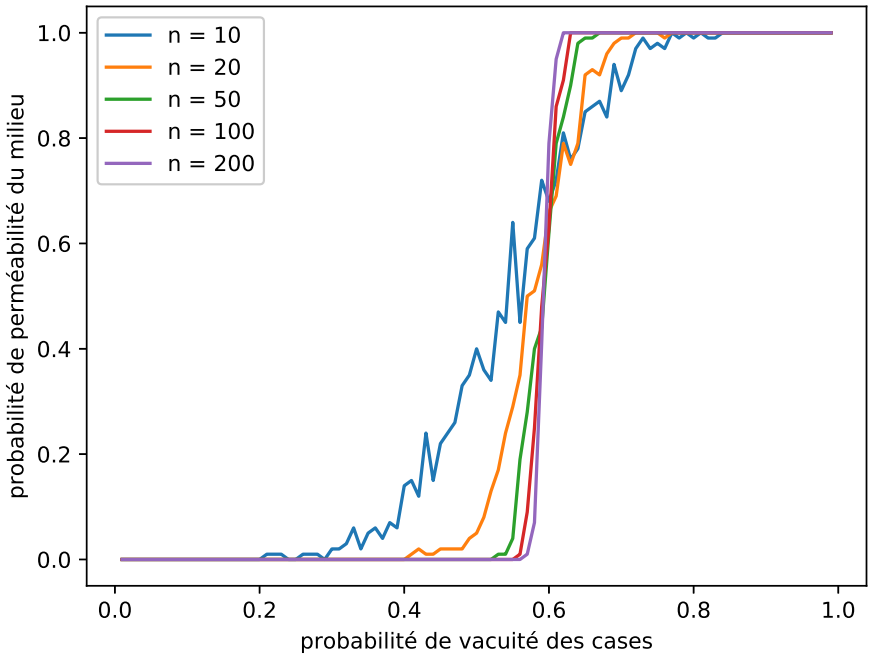
c) Le but est de tracer une courbe montrant la probabilité P que le milieu soit traversé en fonction de la probabilité p qu'une case soit vide. Je choisis donc de renvoyer deux listes destinées à être utilisée par `plot`, la première contenant les valeurs de p considérées, la seconde les valeurs de P estimées en calculant une moyenne sur "plusieurs itérations". Pour plus de souplesse dans l'utilisation de `estimerProba`, j'ajoute à la taille `n` deux paramètres : `nbPts` pour le nombre de points de la subdivision de $[0,1]$ considérés pour p et `nbIter` pour le nombre d'itérations effectuées pour calculer la moyenne servant d'estimation de P.

Pour chaque valeur de p dans ladite subdivision, j'engendre `nbIter` milieux poreux aléatoires dont je teste la perméabilité par un appel à la fonction `parcours` et je compte le nombre de succès.

Sur la page suivante, la fonction `estimerProba` avec un exemple d'utilisation et les courbes obtenues.

```
def estimerProba(n, nbPts = 50, nbIter = 50):
    x, y = [], []
    for i in range(1, nbPts):
        p = i / nbPts
        nbFois = 0
        for k in range(nbIter):
            milieuPoreux = creerMilieu(n, p)
            if parcours(milieuPoreux):
                nbFois += 1
        x.append(p)
        y.append(nbFois / nbIter)
    return x, y

L = [10, 20, 50, 100, 200]
for n in L:
    x, y = estimerProba(n, 100, 100)
    plt.plot(x, y, label = "n = " + str(n))
plt.legend()
plt.xlabel("probabilité de vacuité des cases")
plt.ylabel("probabilité de perméabilité du milieu")
plt.show()
```



d) On voit bien sur les courbes ci-dessus l'effet de seuil annoncé. J'encadre par dichotomie la valeur critique de p, en supposant la probabilité de perméabilité fonction croissante de p, hypothèse raisonnable au vu des courbes, pour n suffisamment grand. Là encore, j'ajoute un paramètre `nbIter` et j'estime la probabilité de perméabilité comme au **c)**.

```
def valCritique(n, eps, nbIter = 200):
    a = 0
    b = 1
    while b-a >= eps :
        m = (a+b)/2
        nbFois = 0
        for k in range(nbIter):
            milieuPoreux = creerMilieu(n, m)
            if parcours(milieuPoreux):
                nbFois += 1
        if nbFois / nbIter < 0.5 :
            a = m
        else :
            b = m
    return (a+b)/2
```

L'appel `valCritique(200, 0.01)` renvoie 0.59.

e) Voici les 3 images obtenues, un peu différentes de celles de l'énoncé, génération aléatoire oblige !