Exercice 1. 1. Compléter la classe `Pile` avec les trois méthodes additionnelles `consulter`, `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 a 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 noté $(1 + 2 \times 3) \times 4$. La valeur d'une telle expression peut être calculée facilement en utilisant une pile pour stocker els 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.

Ecrire une fonction prenant en paramètre une chaîne de caractères 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. Attention : cette fonction ne doit pas renvoyer de résultat si l'expression est mal écrite.

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.

Ecrire 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.

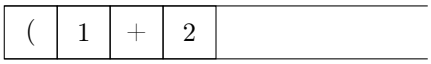
Exercice 4. *Chaînes bien parenthésées.*
On considère une chaîne de caractères incluant à la fois des parenthèses rondes `(` et `)` et des parenthèses carrées `[` et `]`. La chaîne est bien parenthésée si chaque ouvrante est associée à une unique fermante *de même forme*, et réciproquement.

Ecrire 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.

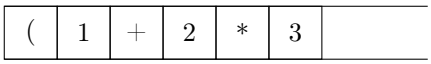
Exercice 5. *Calculatrice ordinaire.*
On souhaite réaliser un programme évaluant une expression arithmétique donnée par une chaîne de caractères. On utilisera les notations et les règles de priorité ordinaires, en supposant pour simplifier que chaque élément est séparé des autres par une espace. Ainsi l'expression $(1 + 2 \times 3) \times 4$ sera décrite par la chaîne de caractères suivante

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

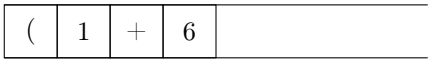
Comme dans l'exercice précédent, nous allons parcourir l'expression de gauche à droite et utiliser une pile. On alterne entre deux opérations : ajouter un nouvel élément sur la pile, et simplifier une opération présente au sommet de la pile. Ainsi dans le traitement de `'( 1 + 2 * 3 ) * 4'`, on ajoute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

Ecrire une fonction `simplifie(pile, ops)` qui simplifie toutes les opérations au sommet de la `pile` utilisant un opérateur de l'ensemble `ops`. En déduire une fonction `calcule(expression)` renvoyant la valeur de l'expression représentée par la chaîne de caractères `expression` prise en paramètre.

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 une telle structure à l'aide d'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 une telle structure à l'aide d'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 cleitn 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 d ela valeur qui vient d'être retirée de la file.
L'idée est de faire tourner une tellesimulation 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 d ela file ou il y a le moins de cleints, 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 N, et on l'affecte à `dispo[i]`. A chaque tour d'horloge, on réalise deux opérations :

- on fait apparaître un nouveau client
- pour chauqe 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]`.

Ecrire 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.