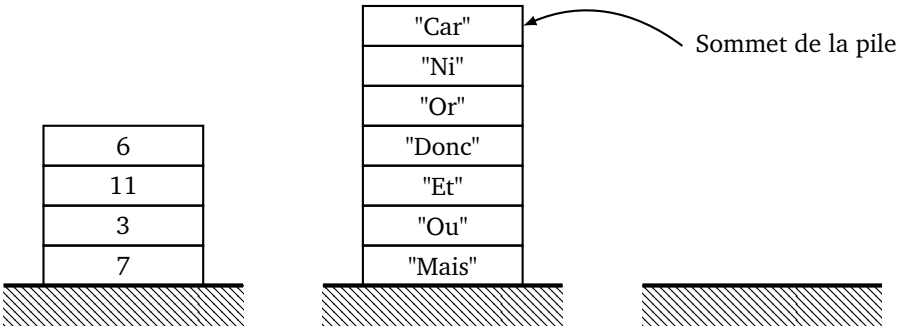
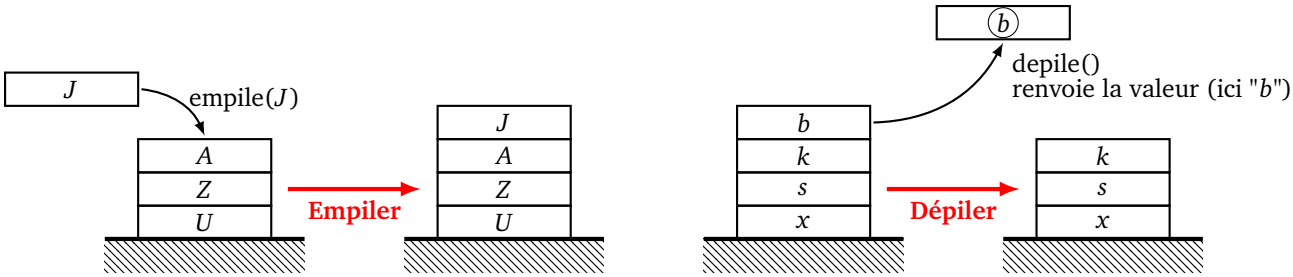
Calculatrice polonaise – Piles

Tu vas programmer ta propre calculatrice ! Pour cela tu vas découvrir une nouvelle notation pour les formules et aussi découvrir ce qu'est une « pile » en informatique.

Cours 1 (Pile)**.**
Une ***pile*** est une suite de données munie de trois opérations de base :

- ***empiler*** : on ajoute un élément au sommet de la pile,
- ***dépiler*** : on lit la valeur de l'élément au sommet de la pile et on retire cet élément de la pile,
- et enfin, on peut tester si la pile est vide.





Remarques.

- **Analogie.** Tu peux faire le lien avec une pile d'assiettes. On peut déposer, une à une, des assiettes sur une pile. On peut retirer, une à une, les assiettes en commençant bien sûr par celle du haut. En plus, il faut considérer que sur chaque assiette est dessinée une donnée (un nombre, un caractère, une chaîne…).
- **Dernier entré, premier sorti.** Dans une file d'attente, le premier qui attend, est le premier qui est servi et ressort. Ici c'est le contraire ! Une pile fonctionne selon le principe « dernier entré, premier sorti ».

- Dans une liste, on peut accéder directement à n'importe quel élément ; dans une pile on n'accède directement qu'à l'élément au sommet de la pile. Pour accéder aux autres éléments, il faut dépiler plusieurs fois.
- L'avantage d'une pile est que c'est une structure de données très simple qui correspond bien à ce qui se passe dans la mémoire d'un ordinateur.

Cours 2 (Variable globale)**.**
Une ***variable globale*** est une variable qui est définie pour l'ensemble du programme. Il n'est généralement pas recommandé d'utiliser de telles variables mais cela peut être utile dans certain cas. Voyons un exemple.
On déclare la variable globale, ici la constante de gravitation, en début de programme comme une variable classique :

```
gravitation = 9.81
```

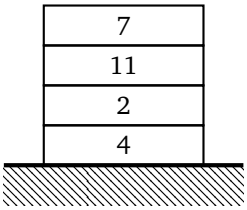
La contenu de la variable `gravitation` est maintenant accessible partout. Par contre, si on souhaite changer la valeur de cette variable dans une fonction, il faut bien préciser à `Python` que l'on est conscient de modifier une variable globale !
Par exemple pour des calculs sur la Lune, il faut changer la constante de gravitation qui y est beaucoup plus faible.

```
def sur_la_lune():
    global gravitation   # Oui, je veux modifier cette variable globale !
    gravitation = 1.625  # Nouvelle valeur pour tout le programme
    ...
```

Activité 1 (Manipulations de la pile)**.**

Objectifs : définir les trois commandes (très simples) pour utiliser les piles.

Dans cette fiche, une pile sera modélisée par une liste. L'élément en fin de liste correspond au sommet de la pile.



Une pile

```
pile = [4,2,11,7]
```

Modèle sous forme de liste

La pile sera stockée dans une variable globale `pile`. Il faut commencer chaque fonction qui modifie la pile par la commande :

```
global pile
```

1. Écris une fonction `empile()` qui ajoute un élément au sommet de la pile.

`empile()`

Usage : `empile(element)`
Entrée : un entier, une chaîne…
Sortie : rien
Action : la pile contient un élément en plus

Exemple : si au départ `pile = [5,1,3]` alors, après l'instruction `empile(8)`, la pile vaut `[5,1,3,8]` et si on continue avec l'instruction `empile(6)`, la pile vaut maintenant `[5,1,3,8,6]`.

2. Écris une fonction `depile()`, sans paramètre, qui retire l'élément au sommet de la pile et renvoie sa valeur.

`depile()`

Usage : `depile()`
Entrée : rien
Sortie : l'élément du sommet de la pile
Action : la pile contient un élément de moins

Exemple : si au départ `pile = [13,4,9]` alors l'instruction `depile()` renvoie la valeur 9 et la pile vaut maintenant `[13,4]` ; si on exécute une nouvelle instruction `depile()`, elle renvoie cette fois la valeur 4 et la pile vaut maintenant `[13]`.

3. Écris une fonction `pile_est_vide()`, sans paramètre, qui teste si la pile est vide ou non.

`pile_est_vide()`

Usage : `pile_est_vide()`
Entrée : rien
Sortie : vrai ou faux
Action : ne fait rien sur la pile

Exemple :
- si `pile = [13,4,9]` alors l'instruction `pile_est_vide()` renvoie `False`,
- si `pile = []` alors l'instruction `pile_est_vide()` renvoie `True`.

Activité 2 (Opérations sur la pile)**.**

Objectifs : manipuler les piles en utilisant seulement les trois fonctions `empile()`*,* `depile()` *et* `pile_est_vide()`*.*

Dans cet exercice, on travaille avec une pile formée d'entiers. Les questions sont indépendantes.

1. (a) En partant d'une pile vide, arrive à une pile `[5,7,2,4]`.

   (b) Exécute ensuite les instructions `depile()`, `empile(8)`, `empile(1)`, `empile(3)`. Que vaut-maintenant la pile ? Que renvoie maintenant l'instruction `depile()` ?

2. Pars d'une pile. Écris une fonction `pile_contient(element)` qui teste si la pile contient un élément donné.
3. Pars d'une pile. Écris une fonction qui calcule la somme des éléments de la pile.
4. Pars d'une pile. Écris une fonction qui renvoie l'avant-dernier élément de la pile (le dernier élément est celui tout en bas ; si cet avant-dernier élément n'existe pas, la fonction renvoie `None`).

Cours 3 (Manipulation de chaînes)**.**

1. La fonction `split()` est une méthode Python qui sépare une chaîne de caractères en morceaux. Si aucun séparateur n'est précisé, le séparateur est le caractère espace.

python : split()

Usage : `chaine.split(separateur)`
Entrée : une chaîne de caractères `chaine` et éventuellement un séparateur `separateur`
Sortie : une liste de chaînes de caractères

Exemple :
- `"Etre ou ne pas etre.".split()` renvoie `['Etre', 'ou', 'ne', 'pas', 'etre.']`
- `"12.5;17.5;18".split(";")` renvoie `['12.5', '17.5', '18']`

2. La fonction `join()` est une méthode Python qui recolle une liste de chaînes en une seule chaîne. C'est l'opération inverse de `split()`.

python : join()

Usage : `separateur.join(liste)`
Entrée : une liste de chaînes de caractères `liste` et un séparateur `separateur`
Sortie : une chaîne de caractères

Exemple :
- `"".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etreounepasetre.'` Il manque les espaces.
- `" ".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etre ou ne pas etre.'` C'est mieux lorsque le séparateur est une espace.
- `"--".join(["Etre", "ou", "ne", "pas", "etre."])` renvoie `'Etre--ou--ne--pas--etre.'`

3. La fonction `isdigit()` est une méthode Python qui teste si une chaîne de caractères ne contient que des chiffres. Cela permet donc de tester si une chaîne correspond à un entier positif. Voici des exemples : `"1789".isdigit()` renvoie `True` ; `"Coucou".isdigit()` renvoie `False`.
   Rappelons que l'on peut convertir une chaîne en un entier par la commande `int(chaine)`. Le petit programme suivant teste si une chaîne peut être convertie en un entier positif :

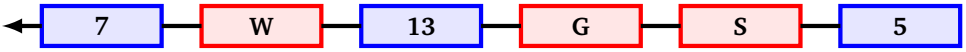
```
machaine = "1789"              # Une chaîne
if machaine.isdigit():
```

```
    monentier = int(chaine)    # monentier est un entier
else:                          # Problème
    print("Je ne sais pas convertir la chaîne en un entier !")
```

Activité 3 (Gare de triage).

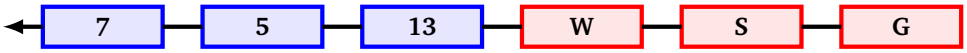
Objectifs : résoudre un problème de triage en modélisant une zone de stockage par la pile.

Un train comporte des wagons bleus qui portent un numéro et des wagons rouges qui portent une lettre.



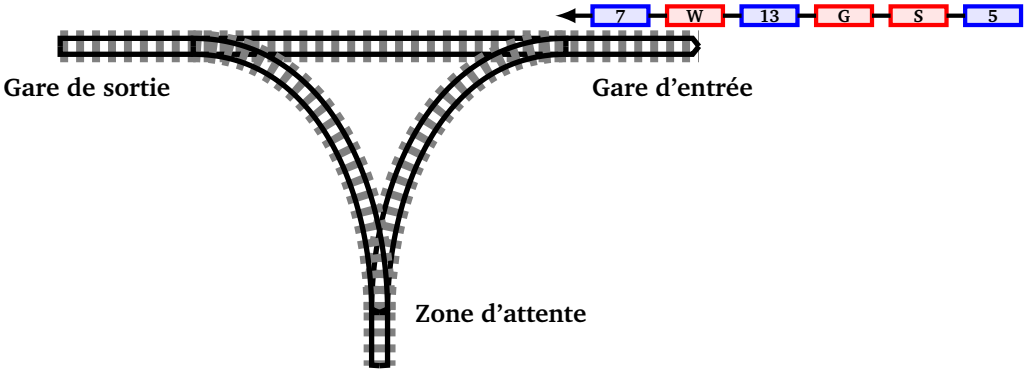
Un train non trié.

Le chef de gare souhaite séparer les wagons : d'abord tous les bleus et ensuite tous les rouges (l'ordre des wagons bleus n'a pas d'importance, l'ordre des wagons rouges non plus).



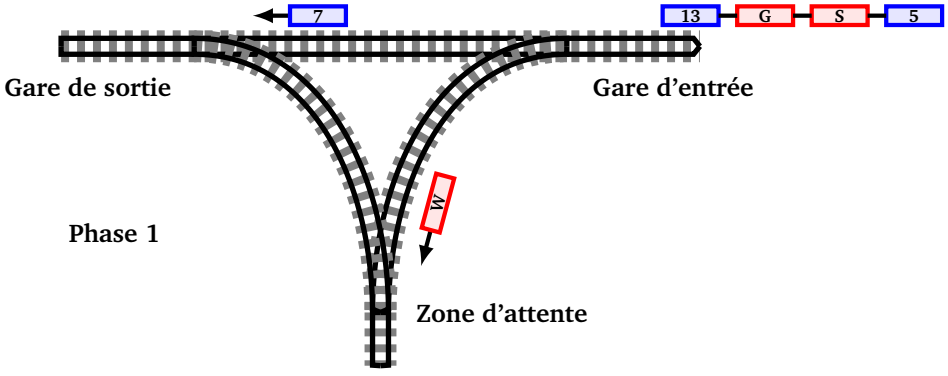
Un train trié.

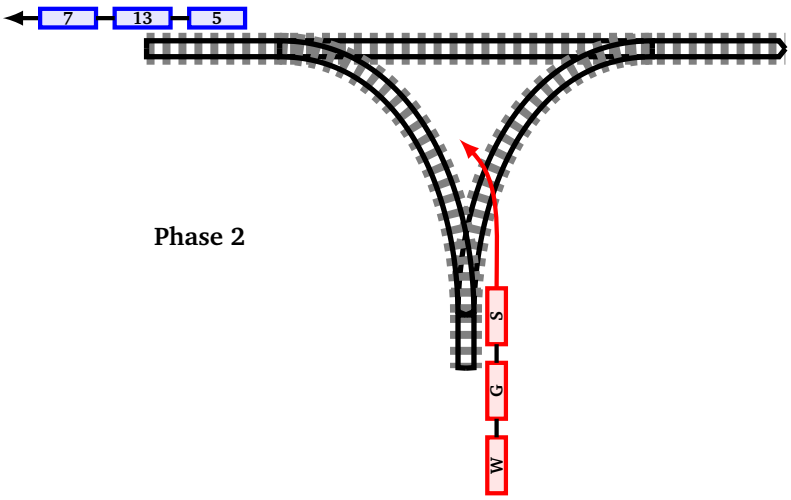
Pour cela, il dispose d'une gare de sortie et d'une zone d'attente : un wagon peut soit être directement envoyé à la gare de sortie, soit être momentanément stocké dans la zone d'attente.



Voici les instructions du chef de gare.

- **Phase 1.** Pour chaque wagon du train :
  — si c'est un wagon bleu, envoyez-le directement en gare de sortie ;
  — si c'est un wagon rouge, envoyez-le dans la zone d'attente.
- **Phase 2.** Ensuite, déplacez un par un les wagons (rouges) de la zone d'attente vers la gare de sortie en les raccrochant aux autres.





Phase 2

Voici comment nous allons modéliser le train et son triage.

- Le train est une chaîne de caractères formée d'une suite de nombres (les wagons bleus) et de lettres (les wagons rouges) séparés par des espaces. Par exemple `train = "G 6 Z J 14"`.
- On obtient la liste des wagons par la commande `train.split()`.
- On teste si un wagon est bleu est regardant s'il est marqué d'un nombre, par la fonction `wagon.isdigit()`.
- Le train reconstitué par les wagons triés est aussi une chaîne de caractères. Au départ, c'est la chaîne vide.
- La zone d'attente sera la pile. Au départ la pile est vide. On va y ajouter uniquement les wagons rouges. À la fin, on vide la pile vers la queue du train reconstitué.

En suivant les instructions du chef de gare, écris une fonction `tri_wagons()` qui sépare les wagons bleus et rouges d'un train.

`tri_wagons()`

Usage : `tri_wagons(train)`
Entrée : une chaîne de caractères avec des wagons bleus (nombres) et des wagons rouges (lettres)
Sortie : les wagons bleus d'abord et les rouges ensuite
Action : utilise la pile

Exemple :

- `tri_wagons("A 4 C 12")` renvoie `"4 12 C A"`
- `tri_wagons("K 8 P 17 L B R 3 10 2 N")` renvoie `"8 17 3 10 2 N R B L P K"`

Cours 4 (Notation polonaise)**.**

L'écriture en notation polonaise (de son vrai nom, notation polonaise inverse) est une autre façon d'écrire une expression algébrique. Son avantage est que cette notation n'utilise pas de parenthèses et qu'elle est plus facile à manipuler pour un ordinateur. Son inconvénient est que nous n'y sommes pas habitués.

Voici la façon classique d'écrire une expression algébrique (à gauche) et son écriture polonaise (à droite).

Dans tous les cas, le résultat sera 13 !

Classique : $7+6$ Polonaise : 7 6 +

Autres exemples :

- classique : $(10+5)\times 3$; polonaise : 10 5 + 3 ×
- classique : $10+2\times 3$; polonaise : 10 2 3 × +
- classique : $(2+8)\times(6+11)$; polonaise : 2 8 + 6 11 + ×

Voyons comment calculer la valeur d'une expression en écriture polonaise.

- On lit l'expression de gauche à droite :

$$2\ 8\ +\ 6\ 11\ +\ \times \longrightarrow$$

- Lorsque l'on rencontre un premier opérateur (+, ×,...) on calcule l'opération *avec les deux membres juste avant cet opérateur* :

$$\underbrace{2\ 8\ +}_{2+8}\ 6\ 11\ +\ \times$$

- On remplace cette opération par le résultat :

$$\underbrace{10}_{\substack{\text{résultat de}\\ 2+8}}\ \ 6\ 11\ +\ \times$$

- On continue la lecture de l'expression (on cherche le premier opérateur et les deux termes juste avant) :

$$10\ \underbrace{6\ 11\ +}_{6+11=17}\ \times \quad \text{devient} \quad 10\ 17\ \times \quad \text{qui vaut} \quad 170$$

- À la fin il ne reste qu'une valeur, c'est le résultat ! (Ici 170.)

Autres exemples :

- 8 2 ÷ 3 × 7 +

$$\underbrace{8\ 2\ \div}_{8\div 2=4}\ 3\ \times\ 7\ +\ \text{devient}\ \underbrace{4\ 3\ \times}_{4\times 3=12}\ 7\ +\ \text{devient}\ 12\ 7\ +\ \text{qui vaut}\ 19$$

- 11 9 4 3 + − ×

$$11\ 9\ \underbrace{4\ 3\ +}_{4+3=7}\ -\ \times\ \text{devient}\ 11\ \underbrace{9\ 7\ -}_{9-7=2}\ \times\ \text{devient}\ 11\ 2\ \times\ \text{qui vaut}\ 22$$

Exercice. Calcule la valeur des expressions :

- 13 5 + 3 ×
- 3 5 7 × +
- 3 5 7 + ×
- 15 5 ÷ 4 12 + ×

Activité 4 (Calculatrice polonaise)**.**

Objectifs : programmer une mini-calculatrice qui calcule les expressions en écriture polonaise.

1. Écris une fonction `operation()` qui calcule la somme ou le produit de deux nombres.

`operation()`

Usage : `operation(a,b,op)`
Entrée : deux nombres `a` et `b`, un caractère d'opération `"+"` ou `"*"`
Sortie : le résultat de l'opération `a + b` ou `a * b`

Exemple :
- `operation(2,4,"+")` renvoie 6
- `operation(2,4,"*")` renvoie 8

2. Programme une calculatrice polonaise, selon l'algorithme suivant :

Algorithme.
- — Entrée : une expression en écriture polonaise (une chaîne de caractères).
  — Sortie : la valeur de cette expression.
  — Exemple : `"2 3 + 4 *"` (le calcul $(2+3)\times 4$) donne 20.
- Partir avec une pile vide.
- Pour chaque élément de l'expression (lue de gauche à droite) :
  — si l'élément est un nombre, alors ajouter ce nombre à la pile,
  — si l'élément est une opération, alors :
    — dépiler une fois pour obtenir un nombre b,
    — dépiler une seconde fois pour obtenir un nombre a,
    — calculer $a+b$ ou $a\times b$ selon l'opération,
    — ajouter ce résultat à la pile.
- À la fin, la pile ne contient qu'un seul élément, c'est le résultat du calcul.

`calculatrice_polonaise()`

Usage : `calculatrice_polonaise(expression)`
Entrée : une expression en notation polonaise (chaîne de caractères)
Sortie : le résultat du calcul
Action : utilise une pile

Exemple :
- `calculatrice_polonaise("2 3 4 + +")` renvoie 9
- `calculatrice_polonaise("2 3 + 5 *")` renvoie 25

Bonus. Modifie ton code pour prendre en charge la soustraction et la division !

Activité 5 (Expression bien parenthésée)**.**

Objectifs : déterminer si les parenthèses d'une expression sont placées de façon cohérente.

Voici des exemples d'expressions bien et mal parenthésées :
- $2+(3+b)\times(5+(a-4))$ est correctement parenthésée ;
- $(a+8)\times 3)+4$ est mal parenthésée : il y a une parenthèse fermante «) » seule ;
- $(b+8/5))+(4$ est mal parenthésée : il y a autant de parenthèses ouvrantes « (» que de parenthèses fermantes «) » mais elles sont mal positionnées.

1. Voici l'algorithme qui décide si les parenthèses d'une expression sont bien placées. La pile joue le rôle d'une zone de stockage intermédiaire pour les parenthèses ouvrantes `"("`. Chaque fois que l'on trouve une parenthèse fermante `")"` dans l'expression on supprime une parenthèse ouvrante de la pile.

   Algorithme.
   Entrée : une expression en écriture habituelle (une chaîne de caractères).
   Sortie : « vrai » si les parenthèses sont cohérentes, « faux » sinon.
   - Partir avec une pile vide.
   - Pour chaque caractère de l'expression lue de gauche à droite :
     — si le caractère n'est ni `"("`, ni `")"` alors ne rien faire !
     — si le caractère est une parenthèse ouvrante `"("` alors ajouter ce caractère à la pile ;
     — si le caractère est une parenthèse fermante `")"` :
       — tester si la pile est vide, si elle vide alors renvoyer « faux » (le programme se termine là, l'expression est mal parenthésée), si la pile n'est pas vide continuer,
       — dépiler une fois, on dépile un `"("`.
   - Si à la fin, la pile est vide alors renvoyer la valeur « vrai », sinon renvoyer « faux ».

   `parentheses_correctes()`

   Usage : `parentheses_correctes(expression)`
   Entrée : une expression (chaîne de caractères)
   Sortie : vrai ou faux selon que les parenthèses sont correctes ou pas
   Action : utilise une pile

   Exemple :
   - `parentheses_correctes("(2+3)*(4+(8/2))")` renvoie `True`
   - `parentheses_correctes("(x+y)*((7+z)")` renvoie `False`

2. Améliore cette fonction pour tester une expression avec des parenthèses et des crochets. Voici une expression cohérente : $[(a+b)*(a-b)]$, voici des expressions non correctes : $[a+b)$, $(a+b]*[a-b)$. Voici l'algorithme à programmer en une fonction `crochets_parentheses_correctes()`.

Algorithme.
Entrée : une expression en écriture habituelle (une chaîne de caractères).
Sortie : « vrai » si les parenthèses et les crochets sont cohérents, « faux » sinon.

- Partir avec une pile vide.
- Pour chaque caractère de l'expression lue de gauche à droite :
  - si le caractère n'est ni `"("`, ni `")"`, ni `"["`, ni `"]"` alors ne rien faire ;
  - si le caractère est une parenthèse ou un crochet ouvrant `"("` ou `"["`, alors ajouter ce caractère à la pile ;
  - si le caractère est une parenthèse ou un crochet fermant `")"` ou `"]"` :
    - tester si la pile est vide, si elle est vide alors renvoyer « faux » (le programme se termine là, l'expression n'est pas cohérente), si la pile n'est pas vide continuer,
    - dépiler une fois, on dépile un `"("` ou un `"["`,
    - si le caractère dépilé (ouvrant) ne correspond pas au caractère lu dans l'expression, alors renvoyer « faux ». Le programme se termine là, l'expression n'est pas cohérente ; dire que les caractères correspondent c'est avoir `"("` avec `")"` et `"["` avec `"]"`.
- Si à la fin, la pile est vide alors renvoyer la valeur « vrai », sinon renvoyer « faux ».

Cette fois la pile peut contenir des parenthèses ouvrantes `"("` ou biens des crochets ouvrants `"["`. Chaque fois que l'on trouve une parenthèse fermante `")"` dans l'expression, il faut que le haut de la pile soit une parenthèse ouvrante `"("`. Chaque fois que l'on trouve un crochet fermant `"]"` dans l'expression, il faut que le haut de la pile soit un crochet ouvrant `"["`.

Activité 6 (Conversion en écriture polonaise)**.**

Objectifs : transformer une expression algébrique classique avec parenthèses en une écriture polonaise. L'algorithme est une version très améliorée de l'activité précédente. Nous ne donnerons pas de justification.

Tu es habitué à l'écriture « $(13+5)\times 7$ » ; tu as vu que l'ordinateur savait facilement calculer « 13 5 + 7 × ». Il ne reste plus qu'à passer de l'expression algébrique classique (avec parenthèses) à l'écriture polonaise (sans parenthèses) !
Voici l'algorithme pour des expressions ne comportant que des additions et des multiplications.

Algorithme.
Entrée : une expression en écriture habituelle
Sortie : l'expression écrite en notation polonaise

- Partir avec une pile vide.
- Partir avec une chaîne vide `polonaise` qui à la fin contiendra le résultat.
- Pour chaque caractère de l'expression (lue de gauche à droite) :
  - si le caractère est un nombre, alors ajouter ce nombre à la chaîne de sortie `polonaise` ;
  - si le caractère est une parenthèse ouvrante `"("`, alors ajouter ce caractère à la pile ;
  - si le caractère est l'opérateur de multiplication `"*"`, alors ajouter ce caractère à la pile ;
  - si le caractère est l'opérateur d'addition `"+"`, alors
    tant que la pile n'est pas vide :
      dépiler un élément,
      si cet élément est l'opérateur de multiplication `"*"`, alors :
        ajouter cet élément à la chaîne de sortie `polonaise`
      sinon :
        empiler cet élément (on le remet sur la pile après l'avoir enlevé)
        terminer immédiatement la boucle « tant que » (avec `break`)
    enfin, ajouter l'opérateur d'addition `"+"` à la pile.
  - si le caractère est une parenthèse fermante `")"`, alors
    tant que la pile n'est pas vide :
      dépiler un élément,
      si cet élément est une parenthèse ouvrante `"("`, alors :
        terminer immédiatement la boucle « tant que » (avec `break`)
      sinon :
        ajouter cet élément à la chaîne de sortie `polonaise`

- Si à la fin, la pile n'est pas vide, alors ajouter chaque élément de la pile à la chaîne de sortie `polonaise`.

`ecriture_polonaise()`

Usage : `ecriture_polonaise(expression)`
Entrée : une expression classique (avec les éléments séparés par des espaces)
Sortie : l'expression en notation polonaise
Action : utilise la pile

Exemple :
- `ecriture_polonaise("2 + 3")` renvoie `"2 3 +"`
- `ecriture_polonaise("4 * ( 2 + 3 )")` renvoie `"4 2 3 + *"`
- `ecriture_polonaise("( 2 + 3 ) * ( 4 + 8 )")` renvoie `"2 3 + 4 8 + *"`

Dans cet algorithme, on appelle abusivement « caractère » d'une expression chaque élément entre deux espaces. Exemple : les caractères de `"( 17 + 10 ) * 3"` sont `(`, 17, `+`, 10, `)`, `*` et 3.
Tu vois que l'addition a un traitement plus compliqué que la multiplication. C'est dû au fait que la multiplication est prioritaire devant l'addition. Par exemple $2+3\times5$ signifie $2+(3\times5)$ et pas $(2+3)\times5$. Si

> tu souhaites prendre en compte la soustraction et la division, il faut faire attention à la non-commutativité ($a - b$ n'est pas égal à $b - a$, $a \div b$ n'est pas égal à $b \div a$).

Termine cette fiche en vérifiant que tout fonctionne correctement avec différentes expressions. Par exemple :

- Définis une expression `exp = "( 17 * ( 2 + 3 ) ) + ( 4 + ( 8 * 5 ) )"`
- Demande à `Python` de calculer cette expression : `eval(exp)`. `Python` renvoie 129.
- Convertis l'expression en écriture polonaise : `ecriture_polonaise(exp)` renvoie
  `"17 2 3 + * 4 8 5 * + +"`
- Avec ta calculatrice calcule le résultat : `calculatrice_polonaise("17 2 3 + * 4 8 5 * + +")` renvoie 129. On obtient bien le même résultat !