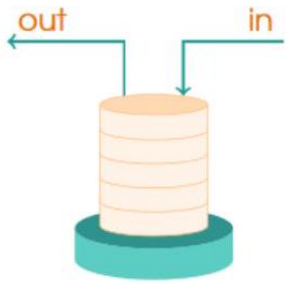
01 - Piles, files et récursivité

1 Les piles

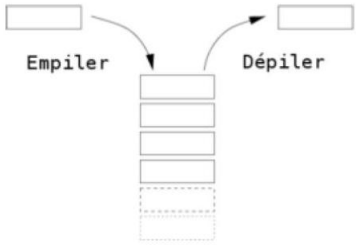
1.1 Définition d'une pile (*stack* in english)



En informatique, une pile est une structure de données fondée sur le principe ≪*dernier arrivé, premier sorti*≫ (ou **LIFO** pour **Last In, First Out**), ce qui veut dire que le dernier élément ajouté à la pile sera le premier à être récupéré.
Le fonctionnement est donc celui d'une pile d'assiettes : on ajoute des assiettes sur la pile,et on les récupère en commençant toujours par la dernière ajoutée.

1.2 Les commandes pour travailler avec une structure de pile

Avant de se pencher sur les différentes façons de réaliser une structure de pile, il faut définir l'ensemble des opérations qu'une telle réalisation doit pouvoir fournir.



Quatre opérations principales devront être définies :

- créer une pile vide
- tester si une pile est vide.
- **empiler** une valeur sur une pile existante.
- **dépiler** une valeur : enlève la valeur du dessus de la pile, et la renvoie.

1.3 Intérêt et inconvénients d'une structure de pile.

Une pile est une structure de données appropriée quand on veut stocker des éléments dont le nombre est variable, et quand on peut se contenter d'accéder au dernier élément stocké.

Cependant, ce ne sera pas adapté si on veut pouvoir accéder à un élément quelconque à tout moment. Il vaudra mieux dans ce cas utiliser une liste ou un tableau (rappel : un tableau est une liste dont tous les éléments sont de même type).

1.4 Exemples d'utilisation d'une pile

- Dans un navigateur Web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton ≪ Afficher la page précédente≫.
- La fonction ≪Annuler la frappe≫ (en anglais ≪ Undo≫, raccourci classique Ctrl+Z) d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

- Les algorithmes récursifs utilisent une pile d'appels, nous allons en parler dans le troisième paragraphe de ce chapitre.
- Les algorithmes de parcours en profondeur d'un graphe utilisent une pile pour stocker les nœuds déjà visités.

1.5 Les piles avec Python

La structure de liste de Python va nous permettre de modéliser une structure de pile, en assimilant le sommet de la pile à la fin de la liste. Les méthodes* **`append`** et **`pop`** vont exactement correspondre aux fonctions empiler et depiler dont nous avons besoin.

(*) Rappel sur ce qu'on appelle une méthode en informatique :

Une méthode est une fonction(que vous pouvez d'ailleurs reconnaître comme telle à la présence des parenthèses) qui est toujours associée à un objet. La syntaxe est **`objet.méthode()`**
Une méthode peut :

- ne pas retourner de valeur mais modifier directement l'objet sur lequel on l'applique.
  Exemple : **`liste.append(5)`** ne retourne rien mais modifie la variable liste
- retourner une valeur en modifiant l'objet sur lequel on l'applique.
  Exemple : **`dernier=liste.pop()`** permet de stocker dans la variable dernier le dernier élément de la liste. De plus liste a été modifiée (le dernier élément a été enlevé)
- retourner une valeur sans modifier l'objet sur lequel on l'applique.
  Exemple : **`nombre='bibi'.index('b')`** permet de connaître le premier indice où se trouve le caractère 'b' dans la chaine 'bibi'. Il est stocké dans la variable nombre.(ici nombre vaut alors.....)

Exercices d'application :

1. Que fait cette fonction **`mystere`** dont l'argument p est une liste (vu comme une pile) ?

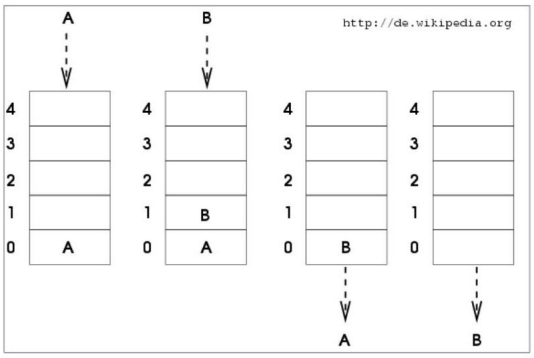
```
1 def mystère(p):
2     a=p.pop()
3     b=p.pop()
4     p.append(a)
5     p.append(b)
```

2. En utilisant **uniquement les méthodes append et pop**, écrire la fonction **`addition(p)`** qui prend en paramètre une pile p d'au moins deux éléments et qui remplace les deux nombres du sommet de cette pile par leur somme. Remarque : cette fonction ne renvoie rien, mais la pile p est modifiée.

2 Les files

2.1 Définition d'une file

Une file est une structure de données fondée sur le principe *"premier arrivé, premier sorti"* (ou **FIFO** pour **First In, First Out**).
Le fonctionnement est donc celui d'une queue à un guichet : la première personne à être arrivée sera la première à sortir de la queue pour être servie.



2.2 Exemple d'utilisation d'une file

- En informatique, les files sont utilisées pour gérer les instructions que doit effectuer un processeur. Les instructions sont stockées sous forme de file dans des buffers d'entrées/sortie, en attente de traitement par le processeur.
- Les algorithmes de parcours en largeur d'un graphe utilisent une file pour stocker les nœuds déjà visités.

2.3 Les files avec Python

Les opérations élémentaires dont on doit disposer pour travailler avec des files sont : créer une file vide, tester si une file est vide, ajouter une valeur à la fin d'une file existante, et extraire une valeur du début de la file.

On voit que si on utilise la structure de liste pour implémenter une file, le principe du FIFO nécessiterait, au moment d'extraire le premier élément, de ré-indexer entièrement tous les éléments de la liste pour tout décaler d'un cran.

Cette opération est coûteuse en temps de calcul, c'est pourquoi il existe un outil spécial pour travailler avec les files, les objets de type **deque**. (Ces objets sont implémentés de façon à ce que les opérations d'ajout et d'extraction soient faites en temps constant)

Remarque : **deque** (prononcer "deck") est l'abréviation de l'anglais *double-ended queue* : il est possible d'ajouter et retirer des éléments par les deux extrémités d'un objet de type deque.

Voici comment construire et manipuler une file avec l'outil **deque** :

```
1 from collections import deque
2 file =deque([ ]) #la file vide
3 file .append('premier')
4 file .append('deuxième')
5 print( file )
6 extrait = file . popleft () #on extrait l'element en tete de file
7 print( extrait )
8 print( file )
```

Exercice d'application :

1. Écrire une fonction nb_occurrence qui prend en paramètres une file **F** (que l'on suppose de type **deque**) et un élément **elt** , et qui renvoie le nombre de fois où **elt** est présent dans la file **F**. **Vous avez uniquement le droit d'utiliser les méthodes append et popleft**, ainsi qu'un test pour savoir si la file est vide *(en particulier il est interdit d'utiliser une boucle qui parcourrait les éléments de la file, et on suppose que l'on n'a pas accès à la longueur de la file)*.
2. Quel est l'état de la file **F** après l'appel de cette fonction ?
3. Si besoin, proposer une version améliorée de la fonction précédente pour qu'après appel de cette fonction, la file **F** ait retrouvé son état d'origine.

3 La récursivité

3.1 Premier exemple : calcul de la factorielle d'un entier

```
1 def  factorielle (n):
2     ''' calcul de n! en utilisant la ré cursivite '''
3     if n==0 :
4         return 1
5     return n* factorielle (n-1)
```

3.2 Principe général de la récursivité

Un algorithme de résolution d'un problème P sur une donnée a est dit récursif si parmi les opérations utilisées pour le résoudre, on trouve une résolution du même problème P sur une donnée b.

Une fonction récursive doit comporter :

- Un cas d'arrêt dans lequel aucun autre appel n'est effectué.
- Un cas général dans lequel un ou plusieurs appels à cette même fonction sont effectués.

Une façon imagée de comprendre la récursivité :
Supposons que vous soyez assis dans un amphithéâtre et que vous souhaitiez vérifier le numéro de votre rangée, que feriez-vous ?
Une possibilité est de demander à la personne devant vous, qui demanderait à la personne devant elle et ainsi de suite jusqu'à ce que la personne de la première rangée soit atteinte. Ensuite, en partant de la première rangée, chaque personne dira son numéro à la personne derrière elle, qui pourra alors connaître son numéro, jusqu'à ce que vous récupériez le vôtre. Voilà, ceci est le principe de base de la récursivité !

3.3 Un inconvénient de la programmation par récursivité

3.3.1 Exemple

Ecrire, sans utiliser de récursivité, une fonction, appelée `factoriellev2` prenant un entier naturel n en argument, et retournant la valeur de $n!$

Lorsque l'on teste les 2 versions de la fonction factorielle pour des valeurs de n de plus en plus grandes, on constate que la version itérative fonctionne très bien, alors que la fonction récursive nous renvoie un message d'erreur lorsque n devient trop grand :

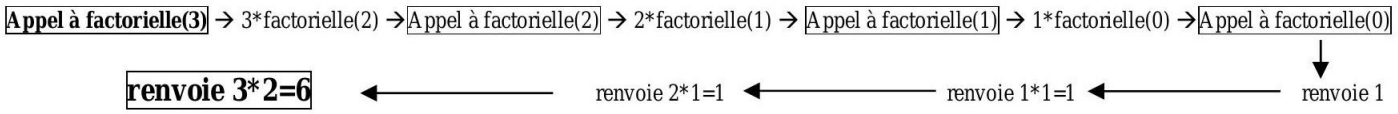
```
RecursionError: maximum recursion depth exceeded in comparison
```

Le problème ici est que le nombre d'appels récursifs est limité par Python (environ 3000 appels récursifs sont acceptés avec les dernières versions), ceci pour éviter une consommation trop grande de l'espace mémoire, comme expliqué ci-dessous.

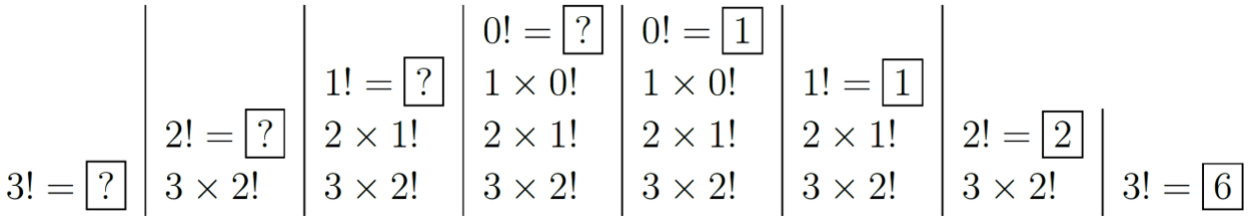
3.3.2 Explication du principe de la récursivité sur l'exemple de la factorielle

Le principe de la récursivité est d'utiliser **une pile d'appels** qui va stocker l'historique des calculs à effectuer.

Schéma pour illustrer l'appel à factorielle(3) :



Le schéma suivant permet de bien visualiser la pile des appels, et la phase de dépilage des calculs :



Plus généralement la récursivité peut être extrêmement longue, il faut être prudent quant à son utilisation.

Nous verrons dans le chapitre suivant des solutions pour améliorer l'efficacité des fonctions récursives, notamment en utilisant un dictionnaire.

4 Les exercices

Exercice 1 : Ecrire une fonction récursive qui calcule la somme des carrés des n premiers entiers naturels, ceci pour tout entier naturel n non nul. (on suppose que l'on ne connaît pas la formule mathématique permettant de calculer directement cette somme)

Exercice 2 : Dans cet exercice, vous n'avez pas le droit à l'instruction Python $**$ pour la puissance (remarque : l'opérateur puissance n'existe pas dans certains langages, par exemple en C)

1. Ecrire une fonction non récursive `puiss_v1(x,n)` de paramètres un réel x et un entier naturel n, qui retourne la valeur de x^n.
2. Ecrire une fonction récursive `puiss_v2(x,n)` de paramètres un réel x et un entier naturel n, qui retourne la valeur de x^n.

Exercice 3 :

1. Ecrire une fonction récursive `inverse(mot)`, d'argument une chaine de caractères, qui renvoie cette chaine écrite à l'envers. Voici le principe :
   - Si cette chaine est de longueur 0, l'inverse de la chaine est elle-même.
   - Pour inverser une chaine de longueur non nulle n, on inverse la chaine formée des $(n-1)$ premiers caractères, que l'on place à la suite du dernier caractère de la chaine initiale.
2. Ecrire une fonction `palindrome(mot)`, qui détecte si ce mot est un palindrome. Cette fonction utilisera la fonction `inverse` écrite dans la question précédente.

Exercice 4 : Proposez une fonction récursive qui compte les occurrences d'une lettre dans un mot. Elle prend deux paramètres, la lettre (de type caractère) et le mot (de type chaine de caractère). Elle retourne le nombre de fois où cette lettre apparaît dans le mot.
Cette fonction exploitera l'idée suivante : si le mot est vide, le nombre d'occurrences est nulle. Sinon on pourra examiner si le premier caractère du mot coïncide avec la lettre cherchée, et lancer la recherche sur les lettres suivantes du mot.

Exercice 5 : *Dans cette exercice, vous utiliserez uniquement les méthodes* **`append`** *et* **`pop`**.
En particulier toute instruction du type "`for x in p`*" ou "*`p[i]`*" est interdite.*

1. Écrire une fonction `renverse(p)` qui prend en entrée une pile p et retourne une autre pile q dont les éléments sont les éléments de p dans l'ordre inverse.
2. Dans quel état se trouve la pile p après un appel à votre fonction `renverse(p)` ?
3. Écrire une fonction `renverse2(p)` qui prend en entrée une pile p et retourne une autre pile q dont les éléments sont les éléments de p dans l'ordre inverse. Par contre on souhaite que la pile p ait retrouvé son état d'origine après l'appel de cette fonction.

Exercice 6 (*Gestion dynamique d'une file de commandes en attente*) : .
Contexte : Vous gérez une file d'attente pour les commandes d'un restaurant en ligne. Les commandes sont ajoutées à la file d'attente au fur et à mesure qu'elles sont reçues. On suppose que chaque commande est associée à un identifiant de commande, et que la file de commande `F` est un objet `deque` qui contient tous les identifiants de commandes. Pour faire cet exercice vous utiliserez uniquement les méthodes `append` et `popleft`.

1. Implémentez une fonction `annuler(F,id)` qui permet de retirer la commande d'identifiant `id` de la file d'attente (par exemple, si le client annule la commande). On suppose que cette commande a bien été passée, et donc son identifiant se trouve quelque part dans la file. Cette fonction ne retourne rien mais doit modifier le contenu de la file `F`.
2. Implémentez une fonction `prioriser(F,id)` qui permet de prioriser la commande d'identifiant `id`, c'est-à-dire la déplacer en tête de la file d'attente. Cette fonction ne retourne rien mais doit modifier le contenu de la file `F`.

Exercice 7 : .
Définition *: le plus grand commun diviseur (ou PGCD) de deux nombres entiers, non tous les deux nuls, est le plus grand entier qui les divise tous les deux. Par exemple, le PGCD de* 20 *et de* 30 *est* 10.
On admet (mais nous serions capable de le démontrer) que si a et b sont deux entiers naturels, non tous les deux nuls, tels que $\mathrm{a} \geq \mathrm{b}$,

$$\mathrm{PGCD}(a, b) = \mathrm{PGCD}(a - b, b)$$

1. (Sur une feuille pour comprendre le procédé) En utilisant plusieurs fois de suite cette propriété, calculer $\mathrm{PGCD}(36, 24)$, puis $\mathrm{PGCD}(16, 9)$. Vous n'avez pas le droit de faire des divisions ou des multiplications. Vous réfléchirez à la condition qui vous a permis, à la fin de votre processus, de conclure.
2. Ecrire une fonction non récursive d'arguments a et b qui calcule le PGCD de a et b.
3. Ecrire une fonction récursive d'arguments a et b qui calcule le PGCD de a et b .

Exercice 8 (*Plus difficile*) : .
Ecrire une fonction récursive d'argument n un nombre entier naturel qui construit la liste de tous les mots binaires de n lettres.
Par exemple `binaire(2)` devra renvoyer la liste `['00','01','10','11']`