Info 2 : Structures de données
Partie I

Contents

Durant cette séance, nous allons étudier différentes structures de données ("objet" contenant des données) les listes, les piles et les files.

1 Les listes

1.1 Usage des listes et quelques révisions

Les listes sont des objets python qui vous sont normalement désormais bien connus. Vérifions malgré tout certaines notions...

Création de listes

Q1. Proposer au moins deux méthodes pour créer une liste contenant tous les entiers de 0 à 9 (inclus).

[2]:
```
L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

L2 = []
for i in range(10):
    L2.append(i)

L3 = [i for i in range(10)]
```

Q2. Quel est l'effet du signe `+` sur les listes ? Tester. Même question pour le signe `*`.

[6]:
```
# Concaténation :
print(L + [10, 11, 12])
```

```
# Multiplication
print(3*L)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5,
6, 7, 8, 9]
```

Q3. En déduire une écriture rapide permettant de créer une liste contenant uniquement des 0, de longeur 30.

[8]:
```
M = [0] * 30
print(M)
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0]
```

Parcours de listes

Pour parcourir et accéder à un ou plusieurs éléments de la liste, il existe différentes méthodes. Considérons la liste suivante :

[10]:
```
L = [2,"chat", True, [0,1], (1,0), "poule"]
```

Q4. Proposer deux méthodes pour afficher un à un tous les éléments de cette liste.

[11]:
```
for i in range(len(L)):
    print(L[i])
```

```
2
chat
True
[0, 1]
(1, 0)
poule
```

[12]:
```
for elem in L:
    print(elem)
```

```
2
chat
True
[0, 1]
(1, 0)
poule
```

On dispose d'une liste de note `Notes` :

[19]:
```
Notes = [randint(5,20) for i in range(10)]
print(Notes)
```

```
[13, 5, 12, 19, 6, 16, 7, 6, 6, 14]
```

Q5. Ecrire un code permettant de donner la moyenne des notes.

[20]:
```
somme = 0
for note in Notes:
```

```
    somme += note
moyenne = somme / len(Notes)
print("Moyenne : ",moyenne)
```

```
Moyenne :  10.4
```

Profitons-en pour introduire un outils qui peut être utile : le débugeur. Nous allons essayer celui de *thonny*. Essayons sur votre précédent code. Pour être encore plus efficace, vérifier si la fenêtre *Variables* est présente (sinon, aller dans le menu *Affichage* puis choisir *Variables*).

Exécuter alors votre code en cliquant sur l'icône *insecte* (*bug* en anglais) ou appuyer sur les touches Ctrl+F5. Seule la première étape du code est alors effectuée. Pour passer à l'étape suivante, cliquer sur l'icône suivante ou appuyer sur F6. On voit ainsi le code "s'éxécuter" avec une pause toute les "grosses" étapes.

Pour être plus précis, on peut utiliser l'icône suivante, ou la touche F7. On voit alors précisément toutes les étapes effectuéees par python. Ces fonxtions peuvent être très utiles pour trouver et comprendre vos erreurs.

A noter que d'autres outils existent : pythontutor, en ligne, et birdseye. Ces deux outils sont disponibles dans le menu *Exécuter*.

On nomme moyenne géométrique de n notes la racine nième du produit des n notes.

Q6. Ecrire un code permettant de calculer la moyenne géométrique des notes.

[22]:
```
produit = 1
for note in Notes :
    produit *= note
moyenne = produit**(1/len(Notes))
print("Moyenne géométriques : ", moyenne)
```

```
Moyenne géométriques :  9.333935625647934
```

Extraction de listes

Q7. Prédire l'action de ces commandes :

[25]:
```
print(L[3])
print(L[2:5])
print(L[:4])
print(L[3:])
print(L[-1])
```

```
[0, 1]
[True, [0, 1], (1, 0)]
[2, 'chat', True, [0, 1]]
[[0, 1], (1, 0), 'poule']
poule
```

1.2 Une remarque importante

Il faut être particulièrement attentif à la copie d'une liste. En effet, pour de *simples* variables :

[2]:
```
a = 3
b = a
b = 2
```

```
print("a :",a,", b :",b)
```

```
a : 3 , b : 2
```

Mais pour des listes :

[4]:
```
L = [1,2,3,4]
M = L
M[2] = 5
print(L)
```

```
[1, 2, 5, 4]
```

Ceci s'explique par le fait que `L` et `M` sont des variables qui pointent en réalité vers la même adresse mémoire : `L` et `M` sont en réalité juste deux façons de nommer le même objet. Modifier `M` modifie les données stockées en mémoire; comme la liste `L` est une adresse vers les mêmes données, elle est aussi modifiée.

Pour éviter ce phénomène, on utilisera la fonction `copy` :

[5]:
```
L = [1,2,3,4]
M = L.copy()
M[2] = 5
print(L)
```

```
[1, 2, 3, 4]
```

Pour la même raison, une fonction modifie la liste surlaquelle elle travaille, même si cette liste n'est pas retournée par la fonction. Par exemple :

[13]:
```
def test_var(v):
    v+=1
    return v
```

Appliquée à la variable `a`, cette fonction retourne une valeur incrémentée de 1. Mais la valeur de `a` n'est pas modifiée :

[20]:
```
a = 2
print("Valeur retournée : ",test_var(a)," / Valeur finale de a : ",a)
```

```
Valeur retournée :  3  / Valeur finale de a :  2
```

Testons avec une liste :

[19]:
```
def test_liste(L):
    for i in range(len(L)):
        L[i]+=1
    return L

M = [1,2,3,4]
print("Liste retournée : ",test_liste(M)," / Liste M : ",M)
```

```
Liste retournée :  [2, 3, 4, 5]  / Liste M :  [2, 3, 4, 5]
```

Ainsi la liste initiale `M` a été modifiée à l'intérieur de la fonction… On pourra même, pour ce type de fonction, se passer du `return ...`, puisque on peut récupérer le résultat dans la liste passée en argument !

A retenir :

- pour garder une copie d'une liste, il faut faire agier la fonction `copy` : `M = L.copy()`
- faire agir une fonction sur une liste modifie cette liste.

Pour aller plus loin... Un comportement similaire peut être obtenu avec les variables "classiques" en les définissant comme globales :

[21]:
```
def test_var_b():
    global a
    a+=1
    return a

a = 2
print("Valeur retournée : ",test_var_b()," / Valeur finale de a : ",a)
```

```
Valeur retournée :  3  / Valeur finale de a :  3
```

Ici la fonction `test_var_b` va chercher la valeur de la variable `a` à l'extérieur de la fonction.

1.3 Mais au fait, qu'est-ce qu'une liste ?

Les listes sont simple d'usage, mais il faut pas perdre de vue que les listes python sont des objets bien mystérieux, et bien plus compliqués qu'espéré, mais c'est aussi ce qui fait une grande partie du succès du langage python ! Comparons ces listes à des objets plus simples :

- Un tuple de python est une structure qui permet de stocker des données de manière similaire aux listes, mais c'est une structure de données non modifiable :

[3]:
```
T=(1,2,3)
T.append(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-af297854506b> in <module>
      1 T=(1,2,3)
----> 2 T.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

Ou bien :

[4]:
```
T[0] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-0e6a27ccfe54> in <module>
----> 1 T[0] = 5

TypeError: 'tuple' object does not support item assignment
```

- Un tableau python, associé à la commande *array* de la bilbiothèque *numpy*, permet aussi de créer un une structure de donnée similaire à une liste. L'avantage de ces structures, c'est qu'elles présentent un accès "rapide" à chaque élément. En effet, de façon schématique, pour accéder au nième élément

d'un tableau, l'ordinateur n'a pas besoin de parcourir les (n-1)ièmes éléments précédents. Ceci est rendu possible car python stocke en mémoire, avec chaque tableau, sa taille : cela permet de savoir directement où trouver chaque élément. On dit que cet élément est accessible en un temps $\mathcal{O}(1)$ pour un tableau, alors que s'il fallait parcourir tous les éléments précédant celui recherché, le temps nécessaire serait $\mathcal{O}(n)$, avec n la taille du tableau. La contrepartie de l'efficacité d'une telle structure, c'est qu'il faut "réserver" en mémoire la place pour le tableau, et cette place réservée dépend de la taille n du tableau : il n'est pas alors possible, facilement (c'est-à-dire rapidement), de changer la taille du tableau, comme le fait la commande *append* pour une liste.

Les listes sont en réalités des objets complexes, s'appuyant sur des tableaux, mais pouvant être redimensionnées *à moindre frais* !

2 Les piles et les files

2.1 Qu'est-ce qu'une file et une pile ?

Une **file** est un ensemble de données qui ne peut être modifié qu'en ajoutant un élément à une extrémité, ou en retirant un à l'**autre** extrémité : c'est le principe du *premier arrivé, premier sorti* ou FIFO en anglais : *First In, First Out.*

Exemples : Toutes les files d'attentes, dans le cas de gens civilisés… Il existe aussi de nombreuses files d'attentes "virtuelles" en informatique (par exemple file d'attentes d'instructions avant traitement par le processeurs, file d'attente de parcoursup, …). Ces structures de données sont aussi intéressantes dans le cadre de la gestion des stocks et denrées périssables.

Une **pile** est un ensemble de données qui ne peut être modifié qu'en ajoutant ou retirant un élément. Ces opérations de modifications ne peuvent avoir lieu qu'à la **même** extremité de la pile : c'est le principe du *dernier arrivé, premier sorti* ou LIFO en anglais : *Last In, First Out.*

Exemples :

- Pile de Caddies au supermarché : les caddies sont rangés les uns derrières les autres. On retire un caddie par la même extremité que celle où on viendra reposer son caddie. Ainsi le caddie qu'on peut sortir est uniquement le dernier inséré !
- Les tours de Hanoï :

Il est ici seulement possible d'ajouter ou de retirer un disque au sommet de chaque tour, qu'on pourrait nommer *pile*.

Une **pile** ou une **file** est une notion théorique, qui possède de multiples réalisations pratiques. Il est possible de les utiliser dans un grand nombre de langage informatique. Pour simplifier, dans la suite, nous étudierons seulement les piles (la manipulation des files repose sur une logique proche, elle pourra être déduite de celle des piles).

2.2 Structure de Pile en python

La structure de pile est une structure qui pourrait tout à fait reposer sur l'objet python "liste" : en effet, il est possible d'ajouter (commande `append`) ou de retirer (commande `pop`) un élement à la fin de la liste :

[23]:
```
L=[1,2,3]
L.append(4)
print(L)
```

```
[1, 2, 3, 4]
```

On a bien ici ajouté un élément à une extremité de la liste, qui constitue une pile. En ré-itérant la commande *append*, on ajoutera toujours un élément à la **même** extremité de cette pile.

Puis, pour retirer cet élément :

[7]:
```
L.pop()
print(L)
```

```
[1, 2, 3]
```

Cependant, dans le cadre pédagogique de cette séance, nous n'utiliserons pas les objets listes pour nos piles : en effet, il est très tentant, avec des piles qui reposent sur des listes, de réaliser des opérations qui sont *interdites* sur de piles (extraire un élément au milieu, ...)

Hors du cadre de ces séances, c'est une pratique réellement employée, pour 2 principales raisons :

- Comme on le dit dans Spiderman, *un grand pouvoir implique de grandes responsabilités* : utiliser des objets complexes (tels que les listes) pour des fonctions simples (empiler ou dépiler) ajoutent des possibilités et effets complexes, parfois inconnus ou non maîtrisés de l'auteur. De façon plus générale, pour des réalisations demandant une *robustesse* extrême, on utilise fréquemment, en informatique, des langages très *basiques*, avec peu de possibilités, afin d'éviter tout effet imprévu.
- Un objet plus simple, devant remplir un cahier des charges plus restreint, peut-être optimisé pour remplir ses (peu nombreuses) missions de façon plus efficace

Nous utiliserons ainsi les structures `deque`, qui sont une généralisation des piles et des files (deque se prononce "dèque" et est l'abréviation de l'anglais *double-ended queue*) : il est possible d'ajouter et retirer des éléments par les deux bouts des deques. Ces opérations sont très efficaces, car les deques sont optimisés pour (plus qu'une liste, qui peut changer de taille, mais de façon plus couteuse en temps d'execution....en contrepartie de ses nombreuses autres fonctions !)

On utilisera pour ces structures *deque* les *méthodes* `append`, `pop` et `len` en employant la même syntaxe qu'avec les listes.

2.3 Créer et utiliser des piles

Pour comprendre le fonctionnement des piles, et les utiliser dans des applications, nous allons créer les fonctions **justes nécessaires** à leur fonctionnement. Ces piles s'appuyeront sur des deque.

Q8. Voici les différentes fonctions essentielles qui nous serons utiles dans la suite, compléter les fonctions lorsque cela est nécessaire et les tester :

[7]:
```
def creer_pile():
    """ Créé et retourne une pile vide """
    return deque()

def taille(P):
    """ Retourne la taille de la pile P """
    return len(P)

def est_vide(P):
    """ Retourne un booléen : True si la pile P est vide, False sinon """
    return taille(P)==0

def empiler(P,v):
    """ Empile l'élément v à la fin de la pile P """
    P.append(v)

def depiler(P):
    """ Dépile la pile P (supprime le dernier élément empilé dans P) """
    if est_vide(P):
        raise ValueError("Erreur : pile vide") ## permet d'afficher un magnifique␣
 ↪message rouge d'alerte si la pile à dépilée est vide
    else :
        P.pop()

def sommet(P):
    """ Retourne le sommet de la pile P """
    if est_vide(P):
        raise ValueError("Erreur : pile vide") ## permet d'afficher un magnifique␣
 ↪message rouge d'alerte si la pile est vide
    else :
        return P[-1]
```

[9]:
```
help(len)
```

```
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

Q9. Programmer la fonction taille_bis(P) uniquement à l'aide des fonctions empiler,depiler et est_vide.

[28]:
```
def taille_bis(P):
    taille = 0
    while est_vide(P) == 0:
        depiler(P)
        taille += 1
    return(taille)
```

Testons le bon fonctionnement :

[30]:
```
P = creer_pile()
empiler(P,1)
empiler(P,4)
taille_bis(P)
```

[30]:
```
2
```

Fonctionne ! Mais attention, la pile est modifiée :

[31]:
```
P
```

[31]:
```
deque([])
```

Q10. Ecrire une fonction `renverser(pile)` prenant en argument une pile et qui retourne la pile obtenue en inversant l'ordre des éléments de `pile`. La pile `pile`pourra être vidée, mais seules les fonctions précédemment définies pourront être utilisées.

[32]:
```
def renverser(pile):
    n = taille(pile)
    Nouv_pile = creer_pile()
    for i in range(n):
        s = sommet(pile)
        depiler(pile)
        empiler(Nouv_pile,s)
    return Nouv_pile
```

[34]:
```
P = creer_pile()
L = ["patates",3,"travers","à","flux","le","est",541]
for elem in L :
    empiler(P,elem)

renverser(P)
```

[34]:
```
deque([541, 'est', 'le', 'flux', 'à', 'travers', 3, 'patates'])
```

2.4 Application directe : vérifier un parenthésage

Le but est de résoudre le problème suivant : on considère une expression mathématique, par exemple : Expression = "1+(3*2+(5/(1+3)))", écrite sous forme d'une chaîne de caractères, et on veut déterminer si elle est bien parenthésée. Cela signifie qu'elle contient autant de parenthèses ouvrantes et fermantes, qui de plus sont bien placées : quand on parcourt la chaîne de gauche à droite, à tout moment, on a rencontré au moins autant de parenthèses ouvrantes que fermantes.

On utilisera bien évidemment la notion de pile et les fonctions définies précédemment.

Q11. Ecrire une fonction Verif_par(Exp) vérifiant si l'expression Exp est bien parenthèsée. Cette fonction retournera une variable booléenne.

[48]:
```
def Verif_par(Exp):
    pile=creer_pile()
    for i in range(len(Exp)):
        if Exp[i]=='(': # Si on rencontre une parenthèse ouverte
```

```
            empiler(pile,i)  # On ajoute dans la pile sa position
        elif Exp[i]==')':# Si on rencontre une parenthèse fermée
            if est_vide(pile)==1: # Si il n'y avait aucune parenthèse déjà ouverte :␣
↪erreur !
                return False
            else :
                depiler(pile) # On "ferme" la dernière parenthèse ouverte
    return est_vide(pile)   # Si il reste des parenthèses ouvertes non fermées ->␣
↪problème, sinon c'est tout bon !
```

[49]:
```
Expression = input("Rentrer l'expression à vérifier ")
print("Bien parenthèsée : ",Verif_par(Expression))
```

```
Rentrer l'expression à vérifier  1+(3*2+5/(1+3)))
```

```
Bien parenthèsée :  False
```

Pour aller plus loin...Q12. Ecrire une fonction Verif_par_bis(Exp) vérifiant si l'expression Exp est bien parenthèsée. Cette fonction retournera une variable booléenne, et l'indice de la parenthèse qui pose problème le cas échéant.

[50]:
```
def Verif_par_bis(Exp):
    pile=creer_pile()
    for i in range(len(Exp)):
        if Exp[i]=='(': # Si on rencontre une parenthèse ouverte
            empiler(pile,i)  # On ajoute dans la pile sa position
        elif Exp[i]==')':# Si on rencontre une parenthèse fermée
            if est_vide(pile)==1: # Si il n'y avait aucune parenthèse déjà ouverte :␣
↪erreur !
                return False,"parenthèse fermée en trop à la position ",i
            else :
                depiler(pile) # On "ferme" la dernière parenthèse ouverte
    if est_vide(pile)==1:
        return True
    else:
        return False, "positions des parenthèses ouvertes non fermées ",pile
```

[51]:
```
Expression =input("Rentrer l'expression à vérifier ")
print("Bien parenthèsée : ",Verif_par_bis(Expression))
```

```
Rentrer l'expression à vérifier  1+(3*2+(5/(1+3)))+3)
```

```
Bien parenthèsée :  (False, 'parenthèse fermée en trop à la position ', 19)
```

2.5 Pour s'éxercer : Tour de Magie

Cet exercice a pour but de modéliser des actions sur un paquet de cartes à jouer. Un paquet de cartes (ou une partie seulement du paquet) peut être modélisé par une pile.

Q13. Couper le jeu de cartes Écrire une fonction `couper(P)` qui prend une pile et qui retire de son sommet `k` éléments un à un qui sont replacés dans une autre pile `Pa`. `k` est un nombre tiré aléatoirement, inférieur au nombre d'éléments de P, grâce à `k = randint(1,taille(P))`. La fonction renvoie `Pa`.

[53]:
```
def couper(P):
    k = randint(1,taille(P))
    Pa = creer_pile()
    for i in range(k):
        s = sommet(P)
        depiler(P)
        empiler(Pa,s)
    return Pa
```

Q14. Battre les cartes On divise le paquet en 2 sous-paquets, afin de le mélanger. Ecrire une fonction `melanger(P1,P2)` qui mélange les éléments des piles `P1` et `P2` dans une troisième pile de la façon suivante : tant qu'aucune pile n'est vide, on retire aléatoirement un élément au sommet d'une des deux piles et on l'empile sur la pile résultat. On utilisera pour cela `randint(1,2)` qui réalise un tirage aléatoire des nombres 1 et 2 (elle retourne 1 ou 2, au hasard). Lorsqu'une pile est vide, on empile le reste de la pile non vide sur la pile résultat.

[54]:
```
def melanger(P1,P2):
    P = creer_pile() # pile résultat
    while est_vide(P1)==False and est_vide(P2)==False : # tant qu'auncune pile n'est␣
 ↪vide
        r = randint(1,2) # on tire au hasard la pile à dépiler
        if r == 1: # si il s'agit de la première
            s = sommet(P1)
            depiler(P1)
            empiler(P,s)
        else : # sinon il s'agit de la deuxième
            s = sommet(P2)
            depiler(P2)
            empiler(P,s)
    while est_vide(P1)==False: # s'il reste des éléments dans la première, on la vide␣
 ↪dans la pile résultat
        s = sommet(P1)
        depiler(P1)
        empiler(P,s)
    while est_vide(P2)==False: #s'il reste des éléments dans la deuxième, on la vide␣
 ↪dans la pile résultat
        s = sommet(P2)
        depiler(P2)
        empiler(P,s)
    return P
```

Q15. Tour de magie de Gilbreath On considère une liste de $n = 5$ cartes (le tour de magie fonctionne quelquesoit le nombre n de cartes considéré).

[55]:
```
L = [9, 10, "reine", "roi", "as"]
```

Ecrire une fonction `magie(L,k)` qui empile k fois cette liste `L` de n cartes dans une pile `P1`(le tour de magie fonctionne quelquesoit le nombre k considéré). Couper alors cette pile avec la fonction `couper`, puis mélanger les deux piles obtenue avec la fonction `melange`, et retourner la pile complète `P`.

On observe alors que la pile finale est constituée de k blocs contenant tous les n cartes de la pile initiale (même si ces dernières peuvent apparaître dans un ordre différent au sein de chaque bloc).

[56]:
```
def magie(L,k):
    P1 = creer_pile()
    for i in range(k):
        for elem in L:
            empiler(P1,elem)
    P2 = couper(P1)
    P = melanger(P1,P2)
    return P
```

Test :

[57]:
```
magie(L,4)
```

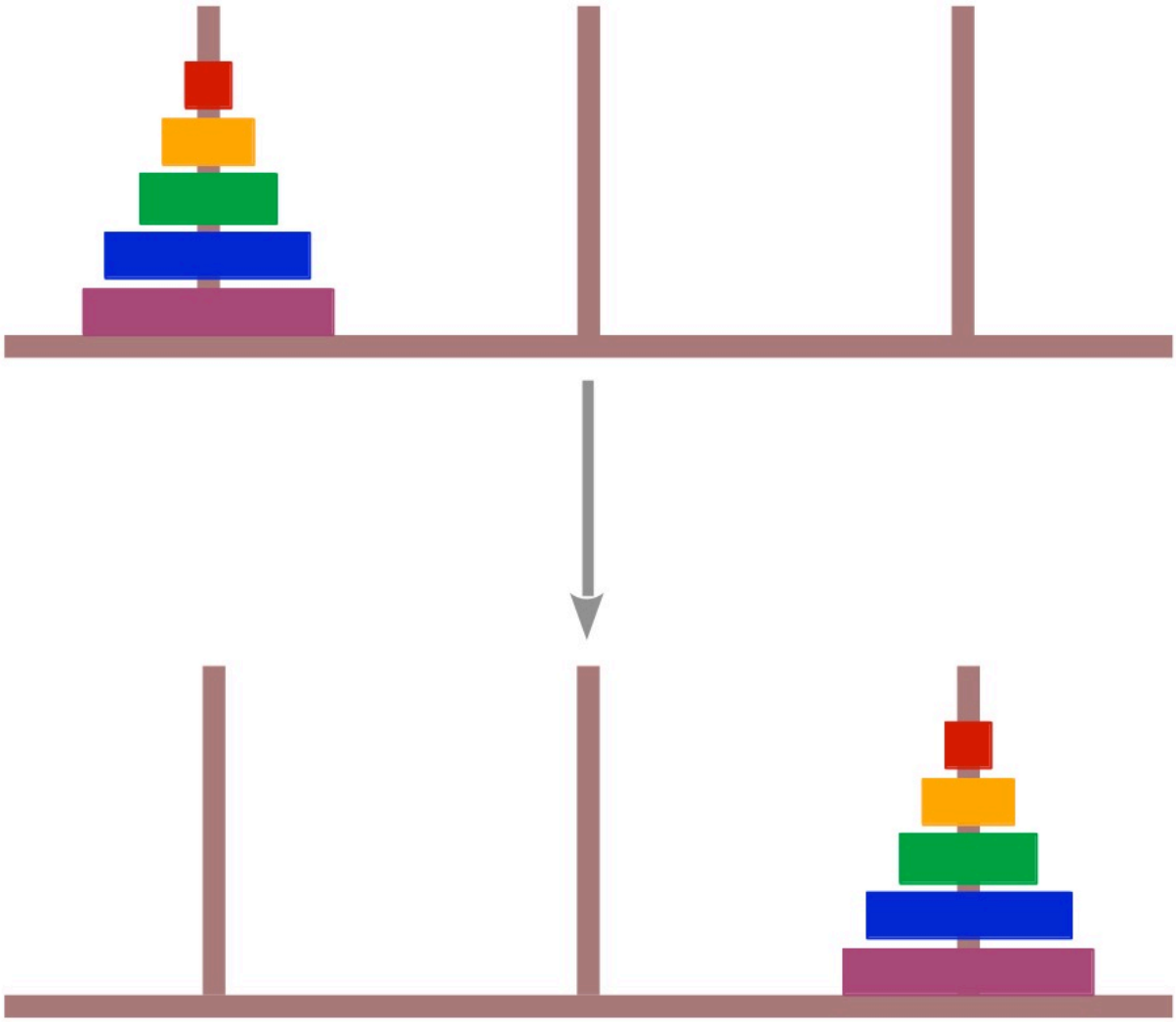
[57]:
```
['roi',
 'as',
 'reine',
 10,
 9,
 9,
 10,
 'reine',
 'roi',
 'as',
 9,
 10,
 'reine',
 'roi',
 'as',
 9,
 10,
 'reine',
 'roi',
 'as']
```

Ca a l'air de fonctionner !

2.6 Pour aller plus loin...Résoudre (bêtement...) le problème des tours de Hanoï

L'objectif de ce casse-tête est de réussir à reproduire sur un autre barreau la tour placée sur le barreau de gauche. Il repose donc sur un principe de piles. En particulier, à chaque déplacement, il n'est autorisé d'empiler sur une pile qu'un disque plus petit que celui au sommet de la pile destinataire. On ne peut déplacer qu'un disque à chaque étape.

Il est possible de s'entraîner à l'adresse suivante : http://jeux.prise2tete.fr/tour-de-hanoi/tour-de-hanoi.php



Q15. En se basant uniquement sur les fonctions créées précédemment, créer les fonctions suivantes :

- creation_tours(n) : Fonction créant l'étant initial. La tour 0 contient n disques dont le dernier est le disque 1 (les disques sont numérotés de 1 à n, dans l'ordre croissant de taille); les tours 1 et 2 sont vides. Cette fonction retourne la liste des 3 piles qu'on appelle « Tours ».
- depiler_tour(Tours) : Fonction qui enlève un disque d'une tour, prise au hasard, et retournant le numéro r de la tour et le numéro s du disque. On utilisera la fonciton randint(a,b) permet de tirer un entier au hasard, entre a et b inclus.
- empiler_tour(Tours,r,s) : Fonction qui place ce disque, le disque s issu de la tour r, sur une nouvelle tour, choisie au hasard, en respectant les règles du jeux !
- resolution(n) : Fonction qui résoud le jeux des tours d'Hanoi, et qui retourne le nombre d'opérations nécessaires.

Voici le préambule du programme à ajouter avant de réaliser le travail ci-dessus :

[8]:
```
from matplotlib.patches import Rectangle

# Fonctions d'affichage en direct - Il suffit d'ajouter en début de simulation et␣
 ↪après chaque déplacement f_animation(Piles) où Piles est la liste des piles. Pour␣
 ↪essayer, taper dans la console: f_animation([[5,4,3,2,1],[],[]])
```

```
def f_init_schema(Pouces):
    fig = plt.figure(1,figsize=(Pouces,Pouces)) # Définition d'une fenêtrenuméro 1 de␣
 ↪taille 20' X 20' dans laquelle l'animation aura lieu
    return fig

def f_affiche_schema(fig,n):
    fig.show() # Affichage du schéma cinématique
    plt.gca().set_aspect('equal', adjustable='box') # L'option Adjustable permet␣
 ↪d'éviter des redimensionnements intempestifs lors de l'évolution de la géométrie
    plt.xlim(-n,5*n)
    plt.ylim(0,n+1)

def f_clear_schema():
    plt.clf() # Effacement de la figure en vue de la prochaine

def f_barreaux(n): # Ajout du segment d'une pièce à la figure du tracé de schéma␣
 ↪cinématique
    for i in range(3):
        X1 = 2*n * i
        Y1 = 0
        X2 = X1
        Y2 = n
        plt.plot([X1,X2],[Y1,Y2],color= "b", linewidth = 10 )

def f_rectangle(fig,x,y,l,h): # Ajout d'un étage à fig en précisant ses coordonnées␣
 ↪(x,y), et ses dimensions (l,h)
    currentAxis = plt.gca()
    currentAxis.add_patch(Rectangle((x,y),l,h,facecolor="grey"))

def f_etage(fig,Pile,Etage,Valeur,n): # Détermination des informations de l'étage␣
 ↪Etage de la pile Pile pour l'ajouter à fig
    Largeur = Valeur
    Hauteur = 1
    X_Med = 2*n * Pile - Largeur / 2
    Y_Med = Etage
    f_rectangle(fig,X_Med,Y_Med,Largeur,Hauteur)

def f_etages(fig,Piles,n): # Ajout de tous les étages de chaque tour à fig
    for i in range(3):
        Pile = Piles[i]
        t = len(Pile)
        for j in range(t):
            Etage = j
            Valeur = Pile[j]
            f_etage(fig,i,Etage,Valeur,n)

def f_animation(LP): # Affichage de la figure avec toutes les informations
    fig = f_init_schema(20)
    f_clear_schema()
```

```
    n = len(LP[0])+len(LP[1])+len(LP[2])
    f_barreaux(n)
    f_etages(fig,LP,n)
    f_affiche_schema(fig,n)
    plt.pause(0.00000000000001)
```

Ces fonctions permettent d'afficher graphiquement les tours : pour cela, taper f_animation(Tours).

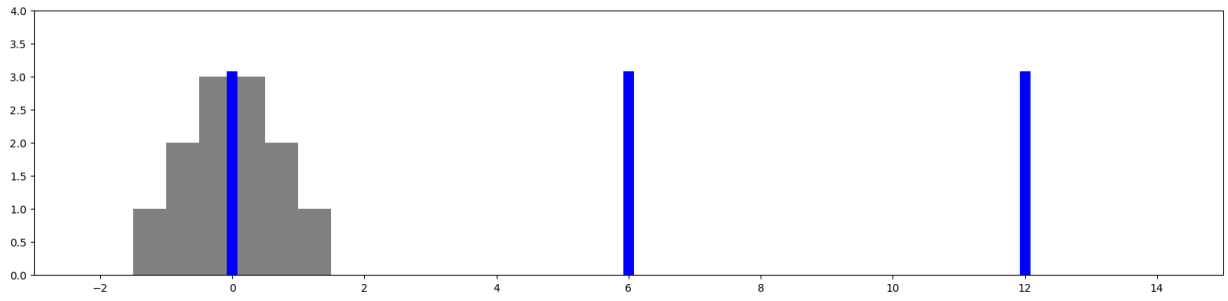
[11]:
```
def creation_tours(n):
    A,B,C=creer_pile(),creer_pile(),creer_pile()
    for i in range(n):
        empiler(A,n-i)
    return [A,B,C]

def depiler_tour(Tours):
    # On commence à choisir la tour à dépiler
    vide=True
    while vide : # on va tirer au sort, mais on évite la ou les tours vides
        r=randint(0,2) # r numéro de la tour tirée au sort
        vide=est_vide(Tours[r])
    # On la dépile en stockant la valeur dépilée
    s=sommet(Tours[r])
    depiler(Tours[r])
    return r,s

def empiler_tour(Tours,r,s):
    # On détermine la tour surlaquelle on va empiler l'élément s
    rn=randint(0,2)
    while est_vide(Tours[rn])==0 and sommet(Tours[rn])<s:
         rn=randint(0,2)# on essaie l'autre tour
    empiler(Tours[rn],s)

def empiler_tour_opt(Tours,r,s):
    # On détermine la tour surlaquelle on va empiler l'élément s
    L=[0,1,2]
    L.remove(r) # pour optimiser le code, petite entorse au règlement : on utilise la␣
 ↪fonction remove
    rn=L[randint(0,1)] # pour optimiser le code, on évite de prendre la même tour que␣
 ↪celle de provenance
    if est_vide(Tours[rn])==0 and sommet(Tours[rn])<s:
        L.remove(rn)
        rn=L[0] # on essaie l'autre tour
        if est_vide(Tours[rn])==0 and sommet(Tours[rn])<s:
            rn=r # en dernier recours, on remet l'élément dans sa colonne d'origine
    empiler(Tours[rn],s)

def resolution(n,aff):
    cpt=0 # compteur du nombre d'opérations
    Tours=creation_tours(n)
    while taille(Tours[2])!=n and taille(Tours[1])!=n :
        r,s=depiler_tour(Tours) # on dépile une tour au hasard
```

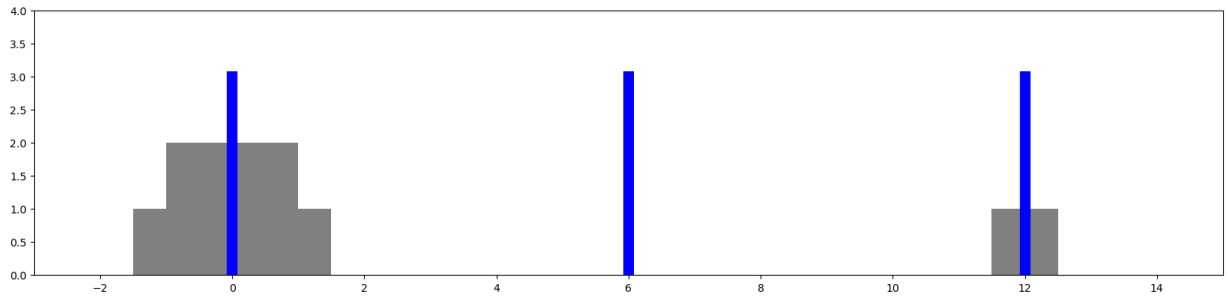
```
        empiler_tour(Tours,r,s) # on empile l'élément dépilé, au hasard
        cpt+=1
        if aff==1 and cpt%10 == 0:
            f_animation(Tours)
    f_animation(Tours)
    return cpt,Tours

def resolution_opt(n,aff):
    cpt=0 # compteur du nombre d'opérations
    Tours=creation_tours(n)
    if aff==1:
            f_animation(Tours)
    while taille(Tours[2])!=n and taille(Tours[1])!=n :
        r,s=depiler_tour(Tours) # on dépile une tour au hasard
        empiler_tour_opt(Tours,r,s) # on empile l'élément dépilé, au hasard
        cpt+=1
        if aff==1 and cpt%10 == 0:
            f_animation(Tours)
    f_animation(Tours)
    return cpt,Tours
```

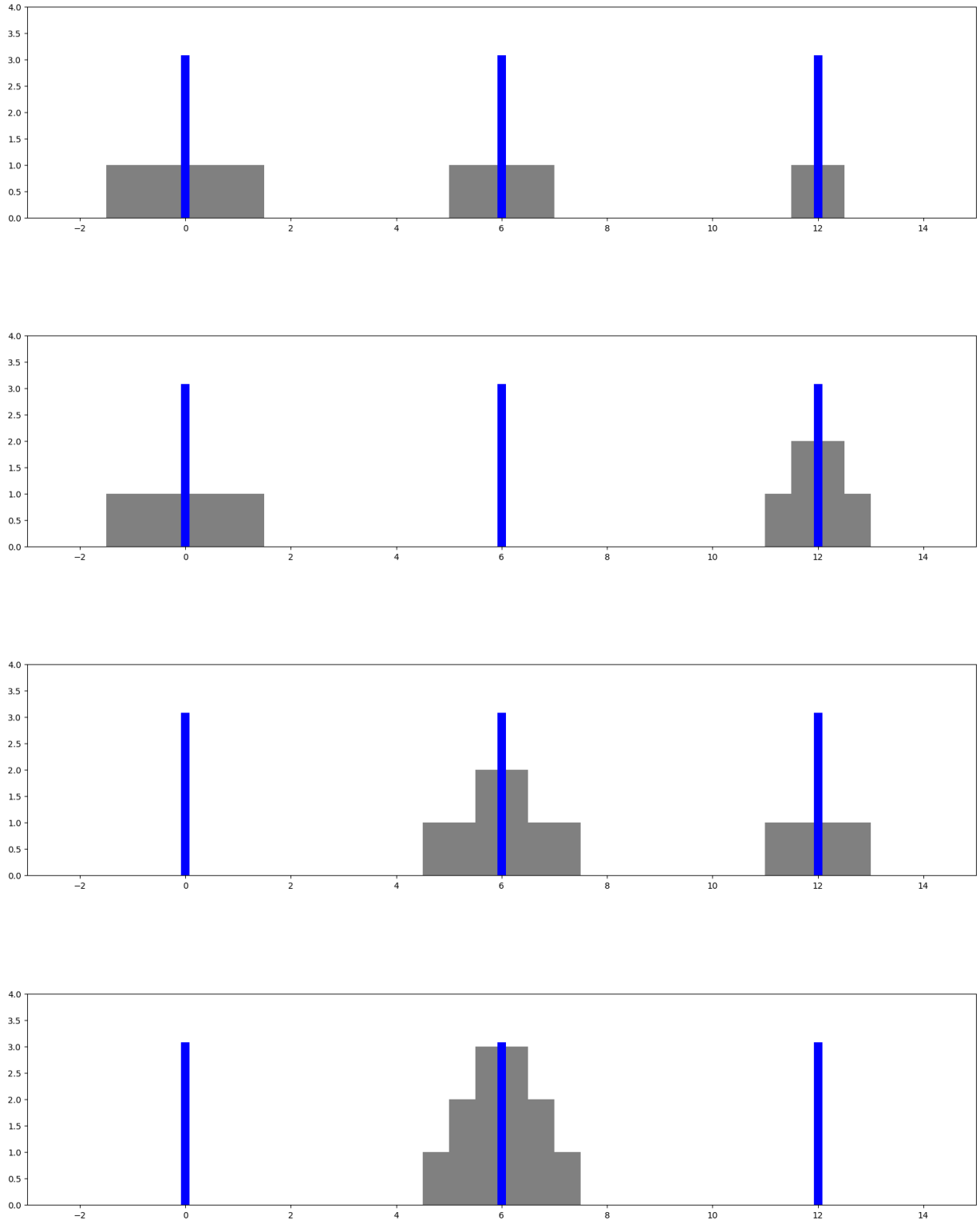
Voici une résolution, avec un affichage de la situation tous les 10 coups :

[13]:
```
resolution_opt(3,1)
```

```
/tmp/ipykernel_326155/2410179758.py:10: UserWarning: Matplotlib is currently
using module://matplotlib_inline.backend_inline, which is a non-GUI backend, so
cannot show the figure.
  fig.show() # Affichage du schéma cinématique
```



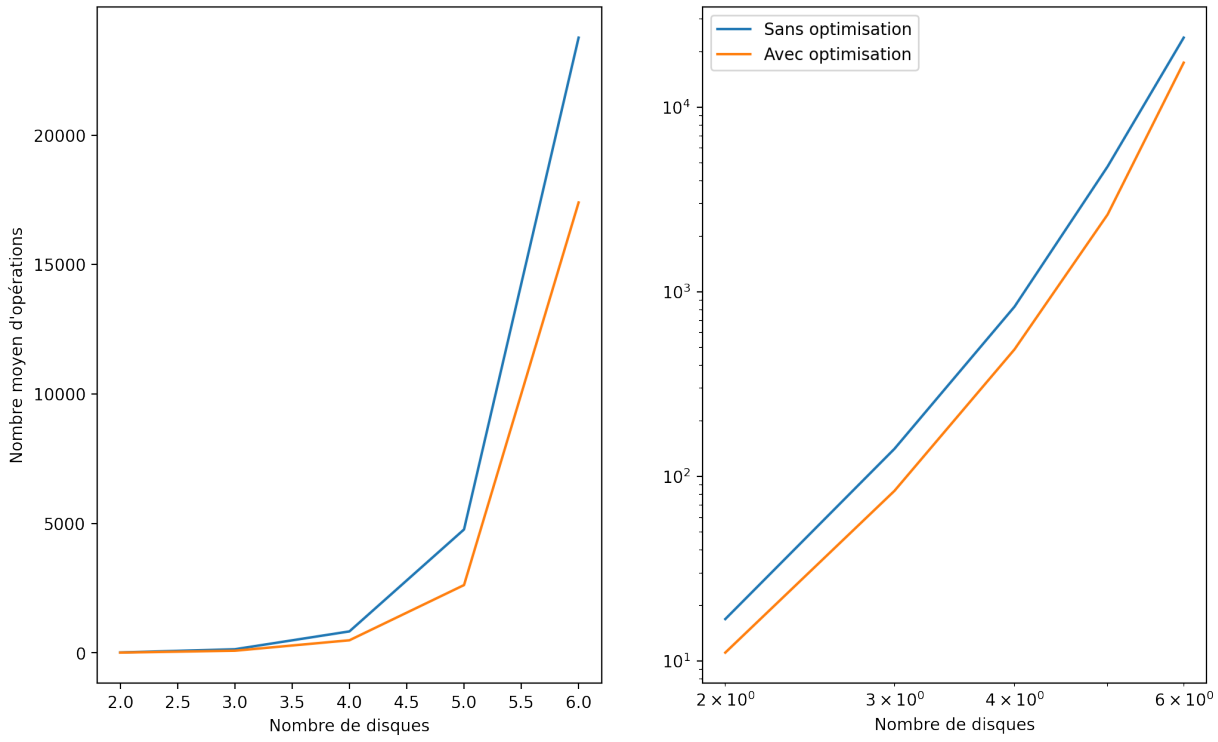




[13]: `(44, [deque([]), deque([3, 2, 1]), deque([])])`

On peut s'interesser aux nombres d'opérations nécessaires en fonction du nombre de disques :

[62]:
```
N,N_opt,K=[],[],[]
for i in range(2,7):
    N.append(sum([resolution(i,0)[0] for k in range(100)])/100) #moyenne sur 100␣
 ↪réalisations
    N_opt.append(sum([resolution_opt(i,0)[0] for k in range(100)])/100)
    K.append(i)

ratio, dpi =  1.5, 200
plt.figure(figsize=(8*ratio,5*ratio),dpi = dpi)
plt.subplot(121)
plt.plot(K,N,label="Sans optimisation")
plt.plot(K,N_opt,label="Avec optimisation")
plt.ylabel("Nombre moyen d'opérations")
plt.xlabel("Nombre de disques")
plt.subplot(122)
plt.plot(K,N,label="Sans optimisation")
plt.plot(K,N_opt,label="Avec optimisation")
plt.yscale('log')
plt.xscale('log')
plt.xlabel("Nombre de disques")
plt.legend()
plt.show()
```



On est bien sûr loin de la solution optimale, permettant de résoudre le problème en 2^n mouvements : http://collection.cassetete.free.fr/1_bois/tour_hanoi/tour_hanoi.htm