



TP – CHAPITRE 3 - Pile, File et liste chaînée

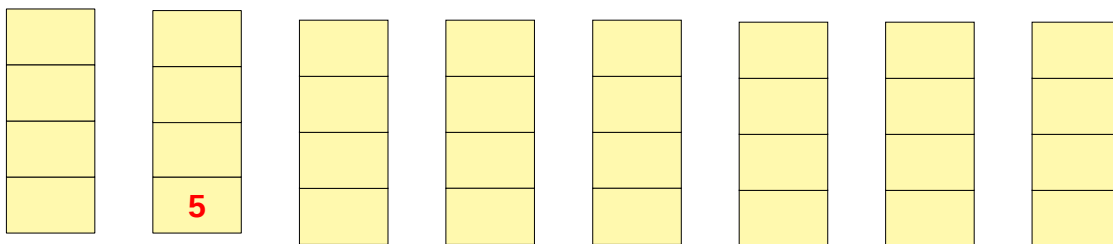


Sommaire

TP1 - Premier exemple de pile.....	2
TP2 - Contrôle du parenthésage d'une expression.....	3
TP3 - Palindrome.....	4
TP4 - Notation polonaise inversée.....	5
TP5 - * Notation polonaise inversée.....	6
TP6 - * Notation polonaise inversée.....	7
TP7 - Premier exemple de file.....	8
TP8 - Nombre de Hamming.....	9
TP9 - Inversion d'une pile.....	10
TP10 - Maximum d'une pile.....	11
TP11 - * Maximum d'une file.....	12
TP12 - Insertion dans une file triée.....	13
TP13 - * Insertion dans une pile triée.....	14
TP14 - Suite de Fibonacci.....	15
TP15 - Implémenter une file avec deux piles.....	16
TP16 - Premier exemple de liste chaînée.....	17
TP17 - Longueur d'une liste chaînée.....	18
TP18 - Placer un nouveau maillon à la fin de la liste chaînée.....	19
TP19 - * Placer un nouveau maillon en nième position.....	19
TP20 - Déterminer si une valeur est dans la liste chaînée.....	20
TP21 - Déterminer la position d'une valeur.....	20

TP1 - Premier exemple de pile

- Sur votre compte, dans le dossier **NSI** , créer un sous-dossier **TP-Chapitre3** dans lequel on rangera les TP de ce chapitre.
- Récupérer les fichiers **pile.py** et **TP1.py** donnés en ressource et les enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Dans le fichier **pile.py** , compléter les **pass** dans la classe **Pile** pour implémenter comme dans le cours une structure de **Pile**.
- Ouvrir le fichier **TP1.py** et exécuter le code. Compléter ensuite les schémas ci-dessous donnant les différentes étapes de l'évolution de la **pile1** correspondant au code de la fonction **exemple()** .



- Le retour attendu dans la console est suivant :

[5, 2, 7]



Lever la main pour valider ce TP.

TP2 - Contrôle du parenthésage d'une expression

- Récupérer le fichier **TP2.py** donné en ressource et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Compléter les **pass** dans la fonction **verifier(chaine)** avec du code afin de vérifier que la chaîne est bien parenthésée, c'est à dire que chaque parenthèse ouvrante est bien fermée plus loin dans la chaîne par une parenthèse fermante.
Pour cela, on va utiliser une pile nommée **pile1** dans laquelle on va empiler les parenthèses ouvrantes quand on rencontre une en parcourant la chaîne de caractère **chaine**.
Et quand on rencontre une parenthèse fermante on dépile la **pile1** si elle n'est pas vide, sinon on renvoie la valeur **False**.
A la fin de cette fonction si la pile est vide c'est que la chaîne est bien parenthésée, sinon elle ne l'est pas.
- Exemple de retours dans la console :

```
>>> %Run TP2.py
Entrer une expression parenthésée :
(((( ))
True

>>> %Run TP2.py
Entrer une expression parenthésée :
(( ))
False

>>> %Run TP2.py
Entrer une expression parenthésée :
(az(e(rt)))p
True
```



Lever la main pour valider ce TP.

TP3 - Palindrome

- Récupérer le fichier **TP3.py** donné en ressource et l'enregistrer dans le dossier **TP-Chapitre2** sur votre compte.
- Au début du code importer la classe **Pile** depuis le fichier **pile.py**.
- Compléter ensuite tous les **pass** dans la suite du code.
Le principe pour vérifier que la chaîne est un palindrome avec une pile, c'est de découper avec des slices la chaîne en deux parties appelées **chaîne1** et **chaîne2**.
On empile ensuite les caractères de la **chaîne1** dans la **pile1**.
Puis on parcourt la **chaîne2** et on compare ses caractères avec ceux que l'on dépile de la **pile1**.
- Exemples de retours dans la console :

```
>>> %Run TP3.py
Entrer une expression :
radar
True
>>> %Run TP3.py
Entrer une expression :
carte
False
```



Lever la main pour valider ce TP.

TP4 - Notation polonaise inversée

Définition : La **notation polonaise inversée (NPI)** est une manière d'écrire les expressions mathématiques en se passant des parenthèses. Elle a été introduite par le mathématicien polonais Jan Lucasiewicz dans les années 1920.

Le principe de cette méthode est de placer chaque opérateur juste après ses deux opérandes.

On utilisera l'**espace** comme séparateur. Par exemple l'expression :

$2 + 3$ devient en NPI : **2 3 +**

$2 * 5 + 3$ devient en NPI : **2 5 * 3 +**

$((1 + 2) * 3) - 4$ devient en NPI : **1 2 + 3 * + -**

$6 + ((2 - 1) * 3)$ devient en NPI : **6 2 1 - 3 * +**

- Ecrire les expressions mathématiques suivantes en **NPI** :

$5 - 6$:

$5 * (7 + 2)$:

$(1 + 3) * (8 - 5)$:

- Le but de ce TP est de réaliser une calculatrice simple, capable d'évaluer une formule en **NPI** et de retourner le résultat arithmétique. La réalisation d'une telle calculatrice se fera à l'aide d'une pile.
L'algorithme est très simple. On commence par lire un par un les caractères de l'expression. Si le caractère lu est un opérande alors on l'empile. Si le caractère lu est un opérateur, alors on dépile les deux éléments se trouvant en haut de la pile, on calcule le résultat en appliquant l'opérateur sur les deux opérandes dépilés et on empile le résultat. Une fois tous les caractères lus, la pile ne contient qu'un seul élément qui correspond au résultat final.

Exemple : Pour l'expression en NPI suivante : **1 2 + 4 * 5 +** les différentes étapes de l'évolution de la pile sont décrites ci-dessous :

		2		4		5	
	1	1	3	3	12	12	17

- Compléter les différentes étapes de l'évolution de la pile pour l'expression : **5 3 - 2 + 7 ***

- Récupérer le fichier **TP4 .py** donné en ressource et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.

- Rajouter du code dans la fonction **evaluer()** pour qu'elle puisse évaluer des expressions en **NPI** avec des opérandes à **un seul chiffre** et les trois opérateurs **+**, **-** et *****.
- Exemple d'affichage dans la console :

```
>>> %Run TP4.py
Entrer une expression en NPI :
2 3 + 1 -
4

>>> %Run TP4.py
Entrer une expression en NPI :
5 3 - 2 + 7 *
28
```



Lever la main pour valider ce TP.

TP5 - * Notation polonaise inversée

- Dupliquer le fichier **TP4.py** en **TP5.py**.
- Modifier le code pour qu'on puisse mettre des opérandes entiers à plusieurs chiffres dans l'expression mathématique en **NPI**.
- Exemple d'affichage dans la console.

```
>>> %Run TP5.py
Entrer une expression en NPI :
15 89 -
-74

>>> %Run TP5.py
Entrer une expression en NPI :
105 98 - 7 *
49
```



Lever la main pour valider ce TP.

TP6 - * Notation polonaise inversée

- Dupliquer le fichier **TP4.py** en **TP5.py**.
- Modifier le code pour qu'on puisse mettre des opérandes décimaux dans l'expression mathématique en **NPI** et qu'on puisse rajouter l'opérateur de division **'/'**.
- Exemple d'affichage dans la console.

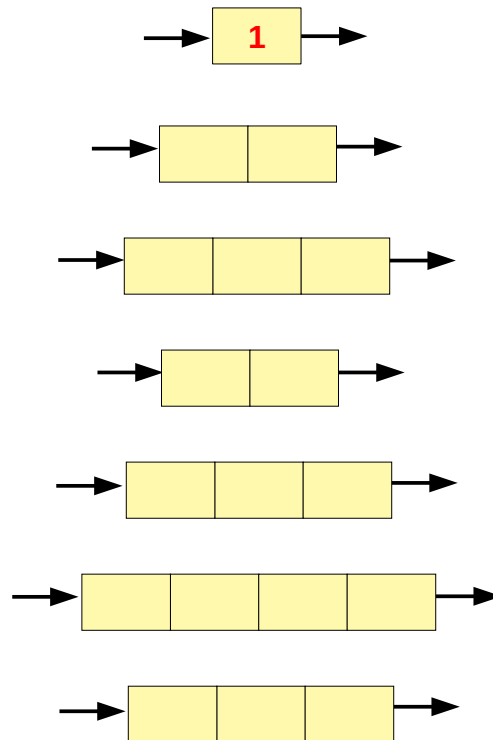
```
>>> %Run TP6.py
Entrer une expression en NPI :
48 5 / 4 -
5.6
>>> %Run TP6.py
Entrer une expression en NPI :
4.5 0.5 - 2 /
2.0
```



Lever la main pour valider ce TP.

TP7 - Premier exemple de file

- Récupérer les fichiers **file.py** et **TP7.py** donnés en ressource et les enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Dans le fichier **file.py**, compléter les **pass** dans la classe **File** pour implémenter comme dans le cours une structure de **File**.
- Ouvrir le fichier **TP7.py** et exécuter le code. Compléter ensuite les schémas ci-dessous donnant les différentes étapes de l'évolution de la **file1** correspondant au code de la fonction **exemple()**.



- Le retour attendu dans la console est suivant :

```
deque([3, 8, 2])
```



Lever la main pour valider ce TP.

TP8 - Nombre de Hamming

Définition : Un nombre de Hamming est un entier naturel qui s'écrit sous la forme : $2^i \times 3^j \times 5^k$ avec i, j et k trois entiers positifs. Autrement dit un nombre de Hamming est un entier naturel pour lequel la factorisation en nombre premier ne fait apparaître aucun nombre premier supérieur à 5.

Le but de ce TP est de rédiger une fonction **hamming(n)** qui retourne la liste des n premiers nombres de Hamming.

Pour cela, on peut utiliser trois structures de **File**, donc implémenter trois files : que l'on nommera **file2**, **file3** et **file5** qui ne contiendront initialement que l'entier 1. Puis on répète n fois le protocole suivant :

- On détermine avec la fonction **min()** le plus petit des trois sommets des trois files, que l'on note **m** et que l'on stocke dans la liste **rep** à retourner à la fin de la fonction et qui sera initialisée à vide au début de la fonction.
- On retire cet élément **m** des files où il se trouve.
- On enfile dans chacune des files : **file2**, **file3**, **file5** respectivement les nombres : $2 * m$; $3 * m$ et $5 * m$.

- Récupérer le fichier **TP8.py** donné en ressource et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Compléter le code de la fonction **hamming(n)** en appliquant le protocole ci-dessus.
- Exemple d'affichage attendu dans la console :

```
>>> hamming(10)
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12]
>>> hamming(20)
[1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36]
```



Lever la main pour valider ce TP.

TP9 - Inversion d'une pile

Pour inverser l'ordre des éléments dans une **pile** on peut utiliser une **file** intermédiaire. En effet, il suffit de dépiler tous les éléments de la pile, de les enfiler dans une file intermédiaire puis de les ré-empiler en les défilant de la file intermédiaire.

- Dans Thonny créer un nouveau fichier nommé **TP9 .py** et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Il faut importer les classes **Pile** et **File** depuis les fichiers **pile.py** et **file.py**.
- Rédiger ensuite une fonction **inversion(p)** qui inverse l'ordre des éléments de la pile **p** en utilisant en variable locale une file **f**.
- Implémenter ensuite une pile nommée **pile1**, la remplir, puis appeler la fonction **inversion**, et faire des affichages dans la console afin d'obtenir l'affichage suivant dans la console.

```
Pile avant inversion :  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
Pile après inversion :  
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

 **Lever la main pour valider ce TP.**

TP10 - Maximum d'une pile

L'objectif de ce TP est de trouver l'élément maximum d'une pile remplie de nombre entier.

- Récupérer le fichier **TP10 .py** donné en ressource et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Rédiger ensuite le code de la fonction **maximum(p)** pour qu'elle retourne la valeur maximum de la pile **p** . Pour cela on peut utiliser une deuxième pile **p1** comme variable locale, dans laquelle on va dépiler la pile **p** pour la parcourir et trouver son maximum. Puis après on rempile la pile **p1** dans la pile **p**.
- Exemple d'affichage attendu dans la console :

```
Contenu de la pile avant recherche :  
[36, 32, 60, 52, 20, 27, 75, 5, 97, 59, 38]
```

```
Maximum de la pile :  
97
```

```
Contenu de la pile après recherche :  
[36, 32, 60, 52, 20, 27, 75, 5, 97, 59, 38]
```



Lever la main pour valider ce TP.

TP11 - * Maximum d'une file

L'objectif de ce TP est de trouver l'élément maximum d'une file remplie de nombre entier.

- Dans Thonny créer un nouveau fichier nommé **TP11 .py** et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Il faut importer la classe **File** depuis le fichier **file.py** et la fonction **randint** depuis le module **random**.
- Rédiger ensuite le code de la fonction **maximum(f)** pour qu'elle retourne la valeur maximum de la file **f**.
- Rédiger ensuite du code pour obtenir dans la console un retour comme ci-dessous :

```
Contenu de la file avant recherche :
```

```
deque([47, 72, 33, 21, 92, 56, 47, 13, 94, 98, 68])
```

```
Maximum de la file :
```

```
98
```

```
Contenu de la file après recherche :
```

```
deque([47, 72, 33, 21, 92, 56, 47, 13, 94, 98, 68])
```



Lever la main pour valider ce TP.

TP12 - Insertion dans une file triée

L'objectif de ce TP est d'insérer un entier dans une file d'entiers triés dans l'ordre décroissant.

- Récupérer le fichier **TP12 .py** donné en ressource et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Rédiger ensuite le code de la fonction **insertion(e, f)** pour qu'elle insère l'entier **e** au bon endroit dans la file **f** qui est triée dans l'ordre décroissant. On peut utiliser une file intermédiaire **f1** comme variable locale.
- Exemple d'affichage attendu dans la console :

```
Contenu de la file avant insertion :
```

```
deque([20, 18, 16, 14, 12, 10, 8, 6, 4, 2, 0])
```

```
Contenu de la file après insertion :
```

```
deque([20, 18, 16, 14, 12, 10, 8, 7, 6, 4, 2, 0, 0])
```



Lever la main pour valider ce TP.

TP13 - * Insertion dans une pile triée

L'objectif de ce TP est d'insérer un entier dans une pile d'entiers triés dans l'ordre décroissant.

- Dans Thonny créer un nouveau fichier nommé **TP13 .py** et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Il faut importer la classe **Pile** depuis le fichier **pile.py** .
- Rédiger ensuite le code de la fonction **insertion(e, p)** pour qu'elle insère un élément **e** dans la pile triée **p** .
- Rédiger ensuite du code pour obtenir dans la console un retour comme ci-dessous :

```
Contenu de la pile avant insertion :  
[20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

```
Contenu de la pile après insertion :  
[20, 18, 16, 14, 12, 10, 8, 7, 6, 4, 2, 2]
```



Lever la main pour valider ce TP.

TP14 - Suite de Fibonacci

L'objectif de ce TP est de rédiger une fonction **fibonacci(n)** qui calcule le nième terme de la suite de Fibonacci en utilisant une pile **p** et une variable **f**.

Pour cela, on suit la procédure suivante :

- étape 1 (initialisation) : On empile **n** dans **p** et on initialise **f** à **0** ;
- étape 2 : (boucle) Tant que la pile **p** n'est pas vide : on dépile **p** ; si le sommet **s** dépilé vaut **0** ou **1**, on ajoute cette valeur à **f**, sinon on empile **s - 1** et **s - 2**.
- étape 3 : On retourne la valeur **f**.

- Dans Thonny créer un nouveau fichier nommé **TP14.py** et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Il faut importer la classe **Pile** depuis le fichier **pile.py**.
- Rédiger ensuite le code de la fonction **fibonacci(n)** en suivant la procédure décrite ci-dessus.
- Rédiger ensuite du code pour obtenir dans la console un retour comme ci-dessous :

```
f(0) = 0
f(1) = 1
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5
f(6) = 8
f(7) = 13
f(8) = 21
f(9) = 34
f(10) = 55
```

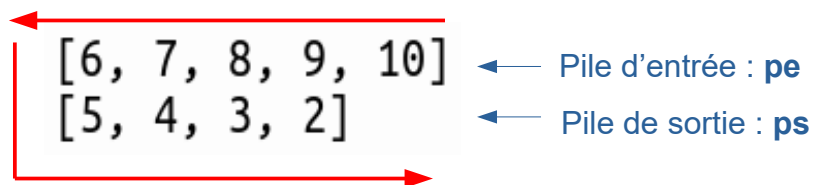


Lever la main pour valider ce TP.

TP15 - Implémenter une file avec deux piles

L'objectif de ce TP est de rédiger une nouvelle classe **File** en utilisant deux structures de **Piles** comme expliqué dans le cours (voir paragraphe 3.4. du chapitre 3).

- Récupérer le fichier **TP15 .py** donné en ressource et l'enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Compléter les **pass** dans la classe **File** pour implémenter comme dans le cours (voir paragraphe 3.4. du chapitre 3) une structure de **File** en utilisant deux **piles** notées **pe** pour pile d'entrée et **ps** pour pile de sortie .
- Le retour attendu dans la console est suivant. La **file** se lit de droite à gauche dans la pile d'entrée **pe**, puis continue de gauche à droite dans la pile de sortie **ps** .



- Dans cet exemple la file est donc : → 10, 9, 8, 7, 6, 5, 4, 3, 2 →



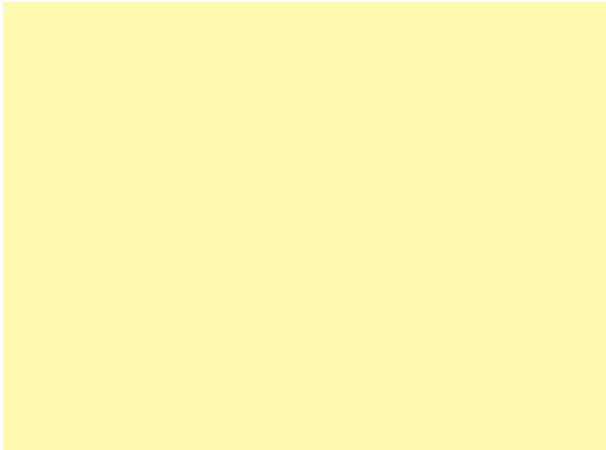
Lever la main pour valider ce TP.

TP16 - Premier exemple de liste chaînée

- Récupérer les fichiers **liste_chaine.py** et **TP16.py** donnés en ressource et les enregistrer dans le dossier **TP-Chapitre3** sur votre compte.
- Dans le fichier **liste_chaine.py**, compléter les **pass** dans les classes **Maillon** et **Liste_chaine** pour implémenter comme dans le cours une structure de **liste chaînée**.
- Ouvrir le fichier **TP16.py** et exécuter le code. Compléter ensuite le tableau ci-dessous donnant les différentes étapes de l'évolution de la **liste chaînée** correspondant au code de la fonction **exemple()**.

Attribut __elements

-> 8



 **Lever la main pour valider ce TP.**

TP17 - Longueur d'une liste chaînée

- Ouvrir le fichier **liste_chaine.py** qui est sur votre compte.
- Dans le fichier **liste_chaine.py**, dans la classe **Liste_chaine** surcharger la fonction **len()** en définissant la méthode **__len__(self)** dans la classe **Liste_chaine** qui retourne la longueur de la liste chaînée, c'est à dire le nombre de valeur dans la liste chaînée.
- Dupliquer le fichier **TP16.py** en **TP17.py** et exécuter le code. Puis vérifier dans la console que l'on obtient le résultat suivant :

```
>>> lc1 = exemple()  
-> 6 -> 4 -> 3 -> 0 -> 8  
>>> len(lc1)  
5
```

- Dans le fichier **TP17.py** modifier le code afin d'obtenir dans la console l'affichage suivant :

```
>>> lc1 = exemple()  
-> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0  
>>> len(lc1)  
10
```



Lever la main pour valider ce TP.

TP18 - Placer un nouveau maillon à la fin de la liste chaînée

- Ouvrir le fichier **liste_chaine.py** qui est sur votre compte.
- Dans le fichier **liste_chaine.py**, dans la classe **Liste_chaine** ajouter une nouvelle méthode **placer_fin(self, v)** qui rajoute un nouveau maillon à la fin de la liste chaînée.
- Récupérer le fichier **TP18.py** donné en ressource et exécuter le code et vérifier qu'on obtient dans la console l'affichage suivant :

```
>>> %Run TP18.py
>>> lc1 = exemple()
      -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
>>> lc1.placer_fin(8)
>>> print(lc1)
      -> 5 -> 4 -> 3 -> 2 -> 1 -> 0 -> 8
```



Lever la main pour valider ce TP.

TP19 - * Placer un nouveau maillon en nième position

- Ouvrir le fichier **liste_chaine.py** qui est sur votre compte.
- Dans le fichier **liste_chaine.py**, dans la classe **Liste_chaine** modifier la méthode **placer(self, v)** en rajoutant un paramètre **n** qui vaut 1 par défaut quand on veut insérer en tête : **placer(self, v, n = 1)**. Modifier ensuite le code de cette méthode pour pouvoir insérer un nouveau maillon dans la liste chaînée en nième position.
- Récupérer le fichier **TP19.py** donné en ressource et exécuter le code et vérifier qu'on obtient dans la console l'affichage suivant :

```
>>> %Run TP19.py
      -> 5 -> 6 -> 7 -> 4 -> 8 -> 3 -> 2 -> 1 -> 0 -> 9
```



Lever la main pour valider ce TP.

TP20 - Déterminer si une valeur est dans la liste chaînée

- Ouvrir le fichier **liste_chaine.py** qui est sur votre compte.
- Dans le fichier **liste_chaine.py**, dans la classe **Liste_chaine** surcharger l'opérateur **in** en définissant la méthode **__contains__(self, valeur)** dans la classe **Liste_chaine** qui retourne un booléen : **True** si valeur est dans la liste chaînée et **False** si ce n'est pas le cas.
- Dupliquer le fichier **TP18.py** en **TP20.py** donné en ressource et exécuter le code et vérifier qu'on obtient dans la console l'affichage suivant :

```
>>> %Run TP20.py
>>> lc1 = exemple()
-> 5 -> 4 -> 3 -> 2 -> 1 -> 0
>>> 3 in lc1
True
>>> 7 in lc1
False
>>>
```



Lever la main pour valider ce TP.

TP21 - Déterminer la position d'une valeur

- Ouvrir le fichier **liste_chaine.py** qui est sur votre compte.
- Dans le fichier **liste_chaine.py**, dans la classe **Liste_chaine** ajouter une nouvelle méthode **index(self, valeur)** qui renvoi la position de la valeur dans la liste chaînée.
- Dupliquer le fichier **TP20.py** en **TP21.py** donné en ressource et exécuter le code et vérifier qu'on obtient dans la console l'affichage suivant :

```
>>> %Run TP21.py
>>> lc1 = exemple()
-> 5 -> 4 -> 3 -> 2 -> 1 -> 0
>>> lc1.index(4)
1
>>> lc1.index(0)
5
```



Lever la main pour valider ce TP.