Tableaux

Pour la représentation des entiers négatifs on a vu la méthode du complément (à 2)
Pour trouver la représentation de -4 **sur un octet** on procède ainsi :

1. On écrit 4 sur un octet -> 0000 0100
2. On prend le complémentaire de 4 -> 1111 1011
3. Puis on ajoute 1 à 1111 1011 et on obtient ainsi la représentation de -4 autrement dit 1111 1100

On aimerait écrire une fonction Python `complementaire(octet)`

1. prenant en entrée un octet, par exemple 0000 0100
2. et retournant le complémentaire de cet octet ici 1111 1011

Comment mémoriser un octet ? Comment mémoriser plusieurs valeurs nommées par un seul nom de variable ?

Un tableau (array en anglais) permet de mémoriser plusieurs valeurs nommées **par un seul nom de variable**

Voici ce que dit la documentation de `numpy`, un module de Python pour le calcul scientifique, à propos des tableaux :

What is an “array”?

In computer programming, an array is a structure for storing and retrieving data. We often talk about an array as if it were a grid in space, with each cell storing one element of the data. For instance, if each element of the data were a number, we might visualize a “one-dimensional” array like a list:

1	5	2	0

A two-dimensional array would be like a table:

1	5	2	0
8	3	6	1
1	7	2	9

Dans les langages comme `C, C++ et Java` il existe des types `array`.

Par contre en `Python` le type proposé pour mémoriser plusieurs valeurs est le type `list` qui est plus riche et plus complexe que le type `array` , donc plus "lent" lorsqu’il faut faire du calcul scientifique avec une grande quantité de données .

Voilà pourquoi le module `numpy` introduit des tableaux pour calculer plus "efficacement " lorsqu’il y a beaucoup de données à traiter.

Most NumPy arrays have some restrictions. For instance:

- All elements of the array must be of the same type of data.
- Once created, the total size of the array can't change.
- The shape must be "rectangular", not "jagged"; e.g., each row of a two-dimensional array must have the same number of columns.

When these conditions are met, NumPy exploits these characteristics to make the array faster, more memory efficient, and more convenient to use than less restrictive data structures.

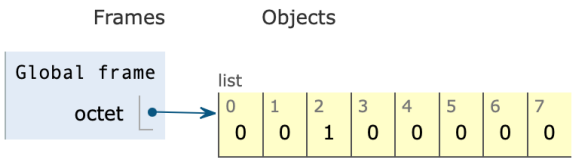
Par la suite nous allons la plupart du temps par souci de simplification utiliser le type `list`.

Ainsi en Python à la variable `octet` on affecte une suite ordonnée de valeurs entre crochets et séparées par des virgules

Or l'écriture 0000 0100 signifie $0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2$ donc on va réécrire les bits de la gauche vers la droite dans l'ordre croissant des puissances de 2

```
octet = [0,0,1,0,0,0,0,0]
```

que l'on visualise ainsi



Ici `octet` n'est qu'un nom utile au programmeur. A ce nom est associée une **référence** (une adresse mémoire) permettant de retrouver toutes les valeurs du tableau.

(Concrètement l'adresse mémoire est celle de **la première valeur du tableau** et puisque on a aussi l'information du nombre d'éléments mémorisés et puisque tous les éléments sont juxtaposés dans la mémoire, on a ainsi accès à tous les éléments du tableau.)

Cette référence est matérialisée par une flèche

Si on veut accéder à une valeur particulière

La première valeur du tableau est repérée par l'indice 0, si on veut accéder à cette valeur et par exemple afficher cette valeur

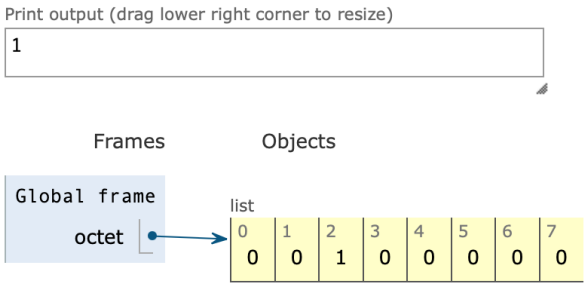
`print(octet[0])` et on obtiendra 0

La deuxième valeur du tableau est repérée par l'indice 1, si on veut accéder à cette valeur et par exemple afficher cette valeur

`print(octet[1])` et on obtiendra 0

La troisième valeur du tableau est repérée par l'indice 2, si on veut accéder à cette valeur et par exemple afficher cette valeur

`print(octet[2])` et on obtiendra 1, etc...



Attention à ne pas chercher une valeur au-delà de la taille du tableau sinon on aura une erreur

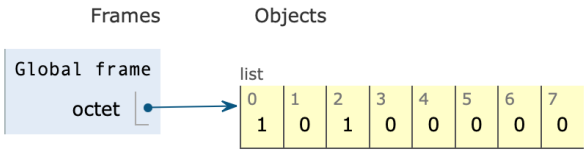
```
>>> octet = [0,1,0,1,0,0,0,0]

>>> octet[8]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: list index out of range
```

On peut modifier la valeur de chaque bit

Par exemple si on veut modifier le bit associé à 2^0 autrement dit le bit repéré par 0 on écrit

```
octet[0] = 1
```

on observe alors



Si on veut connaître la taille du tableau, autrement dit le nombre d'éléments qu'il contient, on dispose pour cela de la fonction `len()` pour **length**.

1 Tableaux et références

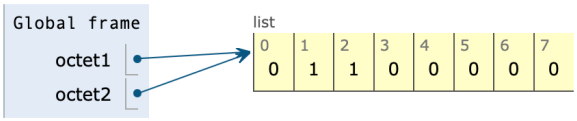
Attention un tableau peut être référencé par plusieurs variables ce qui peut emmener à des situations que l'on n'avait pas forcément prévu

Regardons cela sur un exemple

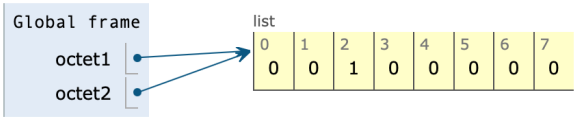
```
octet1 = [0,1,1,0,0,0,0,0]
octet2 = octet1
octet2[1] = 0
```

La variable `octet1` référence les valeurs [0,1,1,0,0,0,0,0]

L'affectation `octet2 = octet1` fait que les deux variables référencent les même valeurs



Par conséquent l'instruction `octet2[1] = 0` change la valeur à l'indice 1 dans le tableau `octet2` **mais aussi dans le tableau** `octet2`



2 Parcours d'un tableau

Pour définir la fonction `complementaire(octet)` il faut une commande permettant d'accéder à toutes les valeurs du tableau,(on dit aussi de parcourir le tableau)

Il existe deux façons de parcourir le tableau au moyen d'une boucle for

1. La première consiste à parcourir tous les indices du tableau de 0 jusqu'à $n-1$ où n est la longueur du tableau

   ```
   for i in range(len(octet)):
       octet[i] = 1 - octet[i]
   ```

   En procédant ainsi on peut modifier la valeur de chaque bit et obtenir le complémentaire

2. La deuxième nous permet **seulement de récupérer la valeur de chaque bit mais ne nous permet pas de modifier la valeur de chaque bit** et par conséquent ne marche pas ici

   Cependant dans des problèmes où on n'aura besoin que de récupérer les valeurs sans modifier les valeurs il est préférable d'utiliser cette méthode pour des raisons de lisibilité du code

   ```
   for bit in octet:
       bit = 1 - bit
   ```

   Avant de définir la fonction regardons dans Python Tutor l'exécution des instructions suivantes :

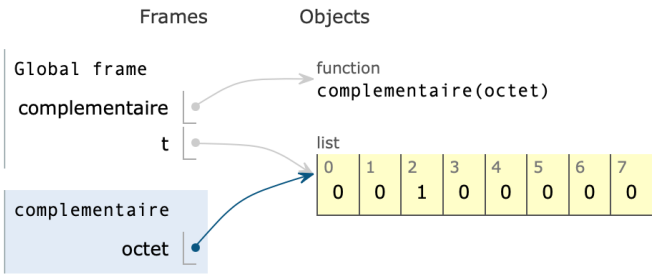
   ```
   octet = [0,0,1,0,0,0,0,0]
   for i in range(len(octet)):
       octet[i] = 1 - octet[i]
   ```

   On observe qu'à la fin de l'exécution le tableau `octet` a pour valeur [1,1,0,1,1,1,1,1]

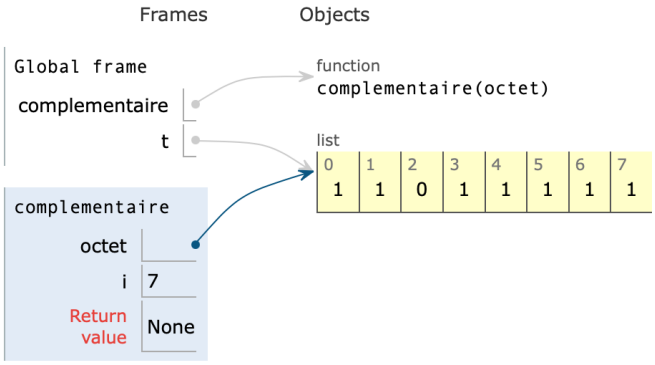
   On définit maintenant la fonction `complementaire(octet)` puis on exécute cette fonction (dans Python Tutor pour visualiser)

   ```
   def complementaire(octet):

       for i in range(len(octet)):
           octet[i] = 1 - octet[i]

   t = [0,0,1,0,0,0,0,0]
   complementaire(t)
   ```

A l'appel de cette fonction le contexte d'exécution de la fonction (boite bleue) contient l'argument `octet` référençant le même tableau que `t`



A la fin de la boucle for le tableau référencé par `octet` et `t` a été modifié

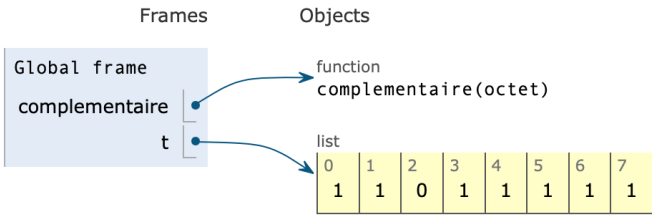


Lorsque les instructions de la fonction ont été toutes exécutées, le contexte d'exécution disparaît et avec lui l'argument `octet` et la variable `i`.

Par contre le tableau a été modifié et on a perdu le tableau du départ.

Nous avons vu un résultat important :

Une fonction peut modifier le contenu d'un tableau passé en argument de cette fonction



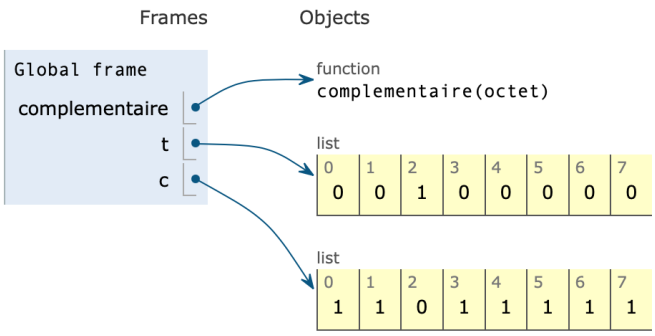
Cependant ceci pose le problème de la perte de la donnée de départ qui a été modifiée.

On aimerait à la fois conserver la donnée de départ et obtenir le complémentaire de l'octet de départ.

Pour cela on va créer un nouveau tableau de même taille **en compréhension**, faire les affectations dans ce nouveau tableau et retourner la référence de ce nouveau tableau.

```
def complementaire(octet):

    comp = [0]*(len(octet))
    for i in range(len(octet)):
```

```
        comp[i] = 1 - octet[i]
    return comp

# Exécution
t = [0,0,1,0,0,0,0,0]
c = complementaire(t)
```



3 Recherche dans un tableau

Un tableau peut servir à mémoriser des documents par exemple

```
documents = ['boucle_for.pdf', 'boucle_while.pdf',..., 'tableaux.pdf']
```

On aimerait savoir si le document `'binaire.pdf'` se trouve dans le tableau `documents`
Une première façon de faire est d'utiliser la commande `in` de Python

```
def appartient(nom,documents):
    return nom in documents
```

Voici un exemple du fonctionnement de `in` en console

```
>>> tab = [1,2,3]

>>> 3 in tab
True

>>> 4 in tab
False
```

Une autre façon est de parcourir le tableau avec une boucle `while` en faisant attention à protéger le booléen `nom != documents[i]` par le booléen `i < len(documents)` pour éviter l'erreur "list index out of range"

```
def appartient(nom,documents):
    i = 0
    while i < len(documents) and nom != documents[i] :
        i += 1
    return i < len(documents)
```

Enfin pour éviter l'erreur "list index out of range" on parcourt le tableau avec une boucle for de laquelle on peut sortir par `return` à condition que l'on ait trouvé la valeur cherchée

```
def appartient(nom,documents):
    for doc in documents:
        if nom == doc:
            return True
    return False
```

4 Tableaux et fonctions

Partie A

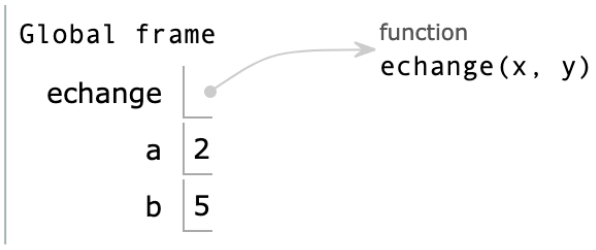
On veut écrire une fonction qui échange les valeurs de deux variables.

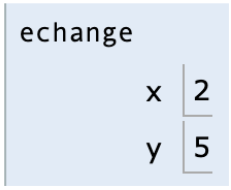
Etudions et observons l'exemple suivant avec Python Tutor.

```
def echange(x:int,y:int)->None:
    temp = x
    x = y
    y = temp

a = 2
b = 5
echange(a,b)
```

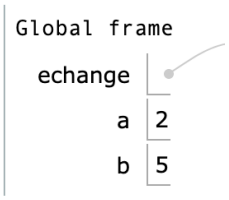
On observe que :

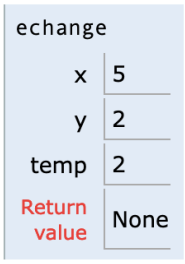
1. Les variables `temp`, `x`,`y` sont **locales** au contexte d'exécution de la fonction (boîte bleue) alors que les variables `a` et `b` sont **globales** (Global Frame)



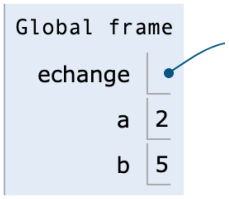


2. L'échange des valeurs se fait dans le contexte d'exécution de la fonction mais ...





3. Une fois que la fonction a été exécutée, le contexte d'exécution "disparaît" ainsi que les variables locales et on constate que les variables globales n'ont pas échangées leur valeur



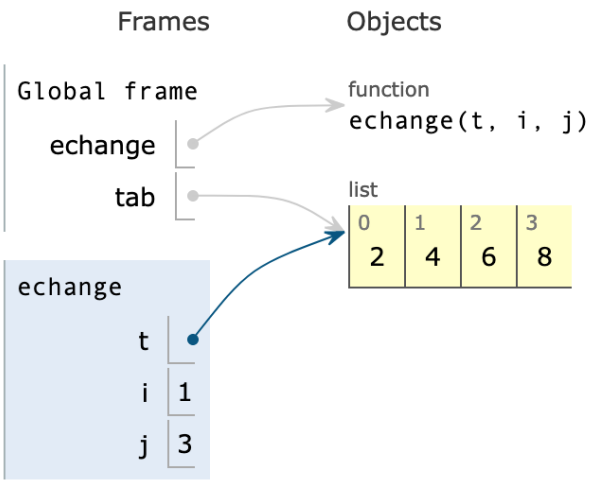
Partie B

On veut écrire une fonction qui échange les valeurs de deux **cellules d'un tableau**. Etudions et observons l'exemple suivant avec Python Tutor.

```
def echange(t:list,i:int,j:int)->None:
    temp = t[i]
    t[i] = t[j]
    t[j] = temp

tab = [2,4,6,8]
echange(tab,1,3)
print(tab)
```

On observe que :

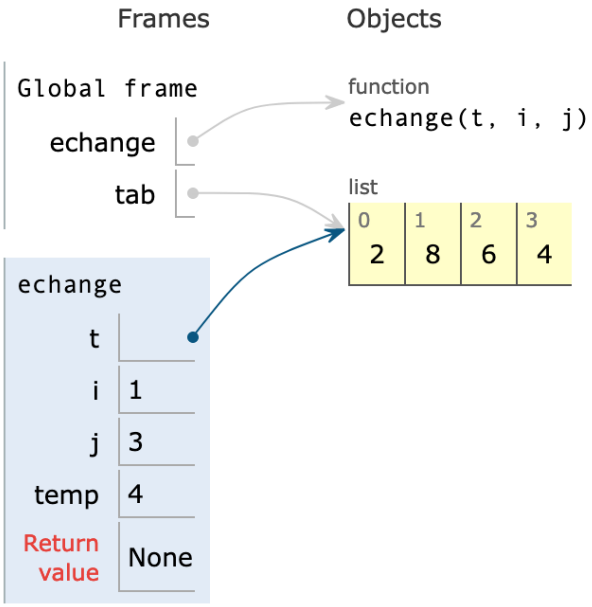
1. Les variables `temp`, `t`,`i` et `j` sont **locales** au contexte d'exécution de la fonction (boîte bleue) alors que la variable `tab` est **globale** (Global Frame)

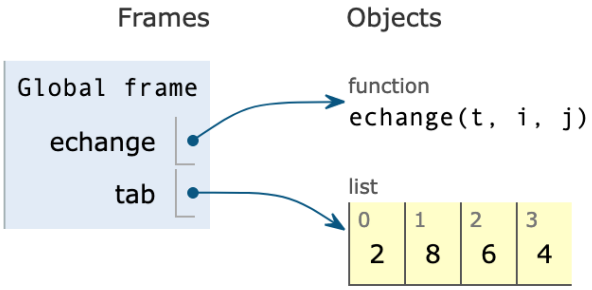


2. L'échange des valeurs pour les cellules d'indice 1 et 3 se fait dans le contexte d'exécution de la fonction.

   On observe que les deux variables `t` locale et `tab` globale font référence à la même zone de la mémoire.

   La modification qui a été faite à partir de l'adresse de `t` concerne donc aussi celle de `tab`, or `tab` est une variable globale donc même si le contexte d'exécution de la fonction disparaît, la modification elle va rester par l'intermédiaire de `tab`





3.

5 Tableaux en compréhension vs tableaux en extension

On dit en Mathématiques, qu'un ensemble est défini ***en extension*** quand on énumère un par un les éléments de cet ensemble.

Par exemple E = {1,2,3,4,5}

Une autre façon est de définir un ensemble ***par compréhension***, c'est à dire en précisant la "logique" qui définit cet ensemble :

Par exemple E est **l'ensemble des carrés des entiers de 1 jusqu'à 20**

Avec un langage de programmation, comme Python on peut aussi définir des tableaux :

1. en extension

   ```
   >>> tab = [1, 2, 3, 4, 5]
   ```

2. par compréhension

   ```
   >>> tab = [i**2 for i in range(1,21)]
   ```

   Un autre exemple :

   On veut les nombres de 0.1 jusqu'à 1 par pas de 0.1 dans un tableau `tab`

   ```
   >>> tab = [0.1*i for i in range(11)]
   ```

   Et si on veut les images des nombres contenus dans `tab` par une fonction mathématique `f` on peut procéder ainsi

   ```
   >>> images = [f(x) for x in tab]
   ```

6 Différences et similitudes entre la classe `list` et la classe `str`

Le type `list` de Python, que l'on a utilisé pour construire des tableaux est **muable**, dans le sens où on peut changer les valeurs des cellules à l'intérieur du tableau.

Listes

Les listes sont des séquences muables, généralement utilisées pour stocker des collections d'éléments homogènes (le degré de similitude varie selon l'usage).

class **list**([*iterable*])

Les listes peuvent être construites de différentes manières :

- en utilisant une paire de crochets pour indiquer une liste vide : `[]` ;
- au moyen de crochets, en séparant les éléments par des virgules : `[a]`, `[a, b, c]` ;
- en utilisant une liste en compréhension : `[x for x in iterable]` ;
- en utilisant le constructeur du type : `list()` ou `list(iterable)`.

On peut ajouter des éléments à une liste Python, par exemple

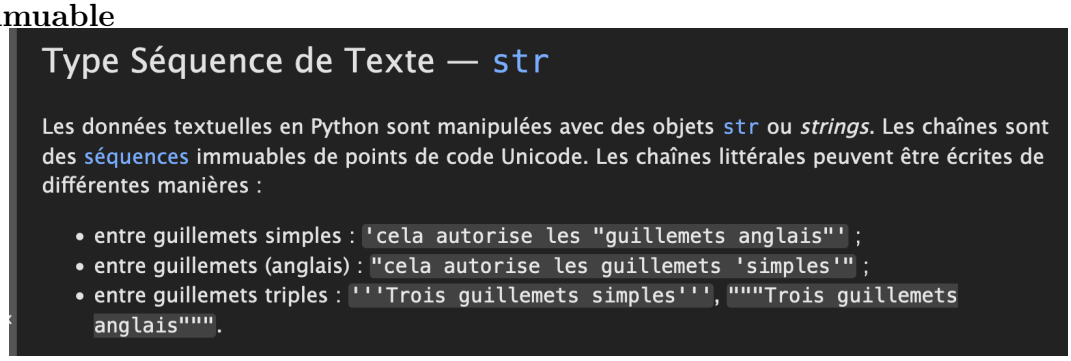
```
>>> liste = []
>>> liste.append(1)
```

```
>>> liste
[1]
>>> liste.append(2)
>>> liste
[1, 2]
```

On peut aussi enlever le dernier élément ajouté à la liste

```
>>> liste.pop()
2
>>> liste
[1]
>>> liste.pop()
1
>>> liste
[]
```

Une chaîne de caractères n'est pas une structure de données, par contre elle est **immuable**



Comme dans un tableau on peut avoir accès à un élément d'indice i entier par exemple

```
>>> mot = 'fonction'
>>> mot[2]
'o'
```

Par contre on ne peut pas modifier un caractère de la chaîne (immuable), si par exemple je veux remplacer la lettre 'o' par a lettre 'u', pour obtenir le mot 'function'

```
>>> mot = 'fonction'

>>> mot[1] = 'u'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

On peut itérer sur les deux classes c'est à dire que l'on peut **parcourir les listes et les chaînes de caractères** afin de résoudre un problème les concernant :

Par exemple dans un **tableau** il n'y a que des caractères 'A', 'T','C' et 'G' et on veut calculer la proportion des lettres 'C' et 'G' par rapport à l'ensemble des lettres

```
def taux_CG(genome:list)->float:
        """"
```

```
        renvoie le taux des lettres C et G
        parmi toutes les lettres contenues
        dans un tableau
        """
        nb_lettres = 0
        for lettre in genome:
            if lettre == 'C' or lettre == 'G':
                nb_lettres += 1
        return nb_lettres/len(genome)
```

Si les lettres sont dans une chaîne de caractères

```
def taux_CG(genome:str)->float:
        """"
        renvoie le taux des lettres C et G
        parmi toutes les lettres contenues
        dans un chaine de caractères
        """
        nb_lettres = 0
        for lettre in genome:
            if lettre == 'C' or lettre == 'G':
                nb_lettres += 1
        return nb_lettres/len(genome)
```

Les fonctions sont quasiment identiques en dehors des annotations et des spécifications. On a privilégié un parcours par valeur, on aurait pu faire un parcours par indice.

On peut concaténer (c'est à dire mettre l'un à la suite de l'autre) deux listes ou deux chaînes de caractères avec le symbole +

Par exemple avec la classe `str`

```
>>> mot1 = 'py'
>>> mot2 = 'thon'
>>> mot1 + mot2
'python'
>>> mot2 + mot1
'thonpy'
```

Maintenant avec la classe `list`

```
>>> liste1 = ['p','y']
>>> liste2 = ['t','h','o','n']
>>> liste1 + liste2
['p','y','t','h','o','n']
>>> liste2 + liste1
['t','h','o','n','p','y']
```

Dans certains problèmes on souhaite passer d'une classe à l'autre.

Par exemple :

1. On a un tableau de caractères et on aimerait faire une chaîne de caractères à partir de tous les caractères.
   On va utiliser pour cela la **méthode** `join` de la classe `str`

   > `str.`**`join`**`(`*`iterable`*`)`
   > Renvoie une chaîne qui est la concaténation des chaînes contenues dans *iterable*. Une `TypeError` est levée si une valeur d'*iterable* n'est pas une chaîne, y compris pour les objets `bytes`. Le séparateur entre les éléments est la chaîne fournissant cette méthode.

   Par exemple :

   ```
   >>> tab = ['G', 'A', 'G']
   >>> mot = '-'.join(tab)
   >>> mot
   'G-A-G'
   ```

   Ou encore

   ```
   >>> tab = ['G', 'A', 'G']
   >>> mot = ''.join(tab)
   >>> mot
   'GAG'
   ```

2. Réciproquement on a une chaîne de caractères et on aimerait obtenir une liste contenant les caractères
   On peut utiliser la fonction `list`

   ```
   >>> mot = 'GAG'
   >>> tab = list(mot)
   >>> tab
   ['G', 'A', 'G']
   ```

   On peut utiliser pour la **méthode** `split` de la classe `str` (lorsqu'il y a un caractère sépérateur)

   > `str.`**`split`**`(`*`sep=None, maxsplit=- 1`*`)`
   > Renvoie une liste des mots de la chaîne, en utilisant *sep* comme séparateur de mots. Si *maxsplit* est donné, c'est le nombre maximum de divisions qui pourront être effectuées (donnant ainsi une liste de longueur `maxsplit+1`). Si *maxsplit* n'est pas fourni, ou vaut `-1`, le nombre de découpes n'est pas limité (toutes les découpes possibles sont faites).
   >
   > Si *sep* est donné, les délimiteurs consécutifs ne sont pas regroupés et ainsi délimitent des chaînes vides (par exemple, `'1,,2'.split(',')` renvoie `['1', '', '2']`). L'argument *sep* peut contenir plusieurs caractères (par exemple, `'1<>2<>3'.split('<>')` renvoie `['1', '2', '3']`). Découper une chaîne vide en spécifiant *sep* renvoie `['']`.

   Par exemple

   ```
   >>> mot = 'G-A-G'
   >>> tab = mot.split('-')
   >>> tab
   ['G', 'A', 'G']
   ```

Pour avoir de l'aide sur la classe `str` en console faire

```
>>> help(str)
```

7 n-uplets (ou tuples an anglais)

Dans certaines situations on ne veut pas que des données stockées puissent changer.

Python propose alors un type **immuable** `tuple` ou n-uplet en français qui permet de stocker une suite de données de telle sorte qu'aucun élément de la suite peut être modifié :

```
>>> num_tel = ('0708091011','0706050403')

>>> type(num_tel)
<class 'tuple'>

>>> num_tel[0] = '0701020304'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Comme les listes on peut concaténer des n-uplets

```
>>> x = ()
>>> y = (1,) # attention (1) n'est pas un tuple mais un entier
>>> z = (2,3)
>>> x + y + z
(1,2,3)
```

On peut affecter une variable de type `tuple` **sans mettre de parenthèses** et en séparant les éléments par une virgule. Les éléments seront alors rassemblées dans un tuple (packing)

Par exemple

```
>>> num_tel = '0708091011', '0706050403'
>>> num_tel
('0708091011', '0706050403')
```

A l'inverse on peut séparer les éléments d'un tuple (unpacking),

Par exemple :

```
>>> n1,n2 = num_tel
>>> n1
'0708091011'
```

Le symbole * écrit devant le nom d'une variable de type `list` ou `tuple` va désassembler la liste ou le tuple et injecter les éléments dans les paramètres d'une fonction

Par exemple au lieu de faire

```
def somme(x,y):
    return x + y
t = (1,2)
a,b = t
print(somme(a,b))
```

On peut faire

```
def somme(x,y):
    return x + y

t = (1,2)
print(somme(*t))
```

On peut aussi utiliser * pour définir une fonction avec un nombre variable d'arguments.

Par exemple, si on veut définir une fonction qui fait la moyenne d'un nombre variables de notes :

On peut procéder ainsi :

```
def moyenne(notes:tuple)->float:
    somme = 0
    for note in notes:
        somme += note
    return somme/len(notes)

print(moyenne((15,12,13,10)))
```

Ou

```
def moyenne(*notes)->float:
    """
    *notes est un ensemble variable de notes
    notes est de type tuple
    """
    somme = 0
    for note in notes:
        somme += note
        compteur += 1
    return somme/len(notes)
print(moyenne(15,12,13,10))
```

On peut utiliser ces deux façons de procéder pour échanger les valeurs de deux variables sans passer par une variable intermédiaire

```
x = 1
y = 2
x,y = y,x
```

Concrètement on a dans l'ordre

1. Créer un tuple t1 avec les éléments y et x (packing)
   `t1 = (y,x)`
2. Affecter aux variables `x` et `y` le contenu de `t1` (unpacking)
   `x,y = t1`

8 n-uplets nommés(ou namedtuples an anglais)

Dans **le traitement de données en tables** certaines données sont caractérisées par des attributs ou des champs.

Par exemple, dans le but d'étudier l'évolution de la température dans une pièce entre le thermostat qui est placé à un endroit de la pièce et le reste de la pièce, on place des capteurs dans plusieurs endroits de la pièce.

Chaque mesure peut avoir plusieurs champs nommés :

1. **x** l'abscisse (on a mis en place un repère, l'origine du repère est le thermostat)
2. **y** l'ordonnée
3. **temp** la température
4. **date** une indication de temps

En Python cela donne :
On crée un objet de type `Mesure` avec quatre attributs `x`, `y`, `temp` et `date`

```
>>> from collections import namedtuple

>>> Mesure = namedtuple('Mesure',['x','y','temp','date'])
        Mesure(x, y, temp, date)
>>> m1 = Mesure()
```

Ensuite on crée une mesure `m1` en donnant des valeurs aux attributs.
On peut rappeler la valeur d'un attribut avec la notation objet.

```
>>> from collections import namedtuple

>>> Mesure = namedtuple('Mesure',['x','y','temp','date'])

>>> m1 = Mesure(10,10,19,200)

>>> m1.temp
19
```

Comme pour les tuples, on peut utiliser l'unpacking et *

```
>>> x,y,temp,date = m1

>>> x
10

>>> print(*m1)
10 10 19 200
```

Exercices

Ex 1

Pour bien comprendre la différence entre le type `array` de `numpy` et `list` de `Python` exécuter les instructions suivantes

1. une liste Python c'est un peu l'auberge "espagnole" on peut ranger des éléments de type différents, ce qui n'est pas le cas pour `numpy`.

   Faire l'import avec un alias `import numpy as np`

```
>>> tab = np.array([1,2,3])

>>> tab[0] = 'pi'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'pi'

>>> liste = [1,2,3]

>>> liste[0] = 'pi'

>>> liste
['pi', 2, 3]
```

2. une liste Python est **dynamique**, sa taille peut changer alors qu'un tableau a une taille fixée une fois pour toute à la création.

```
>>> liste = [1,2,3]

>>> liste.append(4)

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

Ex 2

Voici un programme en python

```
liste1 = [1,1]
liste2 = [0,0]
liste1[0] = 2
liste1 = liste2
print(liste1[0])
```

Qu'observe-t-on à l'écran ? (Justifier par un dessin)

A) 2 B) 1 C) 0

Ex 3

Voici un programme en python

```
liste1 = [0,0,0]
liste2 = [1,1,1]
liste3 = [liste1,liste2]
```

```
liste2 = liste1
print(liste3)
```

Qu'observe-t-on à l'écran ? (Justifier par un dessin)
A) [liste1,liste2] B) [[0,0,0],[1,1,1]] C) [[0,0,0],[0,0,0]]

Ex 4

Le professeur Tournelune range toutes les notes d'un contrôle dans un tableau.

1. Ecrire une fonction `nb_notes(notes,borne_inf)` qui renvoie la proportion en pourcentage des notes contenues dans le tableau `notes` et supérieures ou égales à `borne_inf`
2. Ecrire une fonction `moyenne(notes)` qui renvoie la moyenne de toutes les notes contenues dans le tableau `notes`
3. Ecrire une fonction `reevaluer(notes)` qui **modifie** les notes du tableau `notes` en les augmentant de 10 %
4. Ecrire une fonction `reevaluer_2(notes)` qui renvoie un nouveau tableau contenant les notes réévaluées de 10 %
5. Ecrire une fonction `reevaluer_3(notes)` qui renvoie un nouveau tableau contenant les notes réévaluées de 10 % uniquement si elles sont strictement inférieures à 10

Ex 5

1. Ecrire une fonction python `maximum(tableau)` qui renvoie la plus grande valeur trouvée dans un tableau de nombres `tableau`
   Par exemple `maximum([-2,3,-1,3,1.5])` renvoie 3
2. Ecrire une fonction python `maximum_indice(tableau)` qui renvoie l'indice de la première occurence de la plus grande valeur trouvée dans le tableau de nombres `tableau`
   Par exemple `maximum_indice([-2,3,-1,3,1.5])` retourne l'indice 1 en effet la plus grande valeur 3 se situe aux indices 1 et 3 et on renvoie 1 car c'est le premier indice.

Ex 6

1. Définir une fonction `echange(tableau,i,j)` qui échange dans le tableau les éléments d'indice i et j
2. Se servir de la fonction précédente pour définir une fonction `inverser(tableau)` qui **modifie** le tableau `tableau` en inversant l'ordre des éléments du tableau
   Ainsi à la console on obtient

   ```
   >>> tableau = [1,2,3]
   >>> inverser(tableau)
   >>> tableau
   [3,2,1]
   ```

3. Faire la même chose sans modifier le tableau de départ.
4. Définir une fonction `echange(x,y)` qui échange les valeurs des variables `x` et `y`

Ex 7

Pour mesurer les différences entre deux tableaux de même taille contenant des entiers, on compte **le nombre de différences entre les éléments ayant même indice**

Par exemple si `tableau1 = [1,2,3]` et `tableau2 = [1,3,2]` alors :

1. `tableau1[0]` et `tableau2[0]` sont identiques
2. `tableau1[1]` et `tableau2[1]` sont différents, de même `tableau1[2]` et `tableau2[2]` sont différents donc il y a deux différences entre les deux tableaux.

Définir une fonction `distance(tableau1,tableau2)` qui renvoie le nombre de différences entre les deux tableaux

Par exemple

```
>>>tableau1 = [1,2,3]
>>>tableau2 = [1,3,2]
>>>distance(tableau1,tableau2)
2
```

Ex 8

On définit une suite d'entiers (appelée suite de Fibonnacci) de la manière suivante :

1. Initialisation : les deux premières valeurs sont dans l'ordre 0 et 1
2. Récurrence : chaque valeur suivante est le résultat de la somme des deux valeurs précédentes

Ecrire une fonction `suite_fibonnacci(n)` qui renvoie le tableau des `n` premières valeurs de la suite de Fibonnacci

Ex 9

La suite de Fibonnacci a de nombreuses propriétés :

Vérifier pour des valeurs de `n` relativement grandes que :

1. Si on ajoute les n premiers termes de la suite on obtient le n+2 ième terme -1
2. Si on élève le n ième terme au carré alors on obtient le produit du terme qui précède et du terme qui succède -1 si n est pair et + 1 sinon.

Ex 10

On lance un dé à six faces **tant que les chiffres de 1 à 6 ne sont pas tous sortis**

On aimerait connaître le nombre de lancers moyen pour que les six chiffres apparaissent

1. Engendrer en compréhension une liste `chiffres_sortis` de 7 booléens False.

   Lorsque le 2 sort on fait `chiffres_sortis[2] = True`
2. Définir une fonction Python `globalement_vrai(tableau)` qui renvoie Vraie si le tableau de booléens `tableau` ne contient que des `True`
3. Définir une fonction `nb_lancers()` qui renvoie le nombre de lancers de dé effectués pour faire apparaître tous les chiffres.
4. Définir une fonction `nb_lancers_moyen(nb_repetitions)` qui renvoie le nombre de lancers moyen de dé effectués pour faire apparaître tous les chiffres.

Ex 11

Prolonger l'exercice précédent

Problème : Y-a-t-il une relation entre le nombre de faces d'un dé et le nombre moyen de lancers nécessaire pour faire apparaître tous les nombres sur les faces ?

Ex 12

Etant donné un tableau d'entiers `t` on définit le tableau `diff` des différences de `t` par `diff[i] = t[i+1] - t[i]` pour `i` variant de 0 jusqu'à `len(t) - 1`

1. Ecrire une fonction `difference(tableau)` qui renvoie le tableau des différences de `tableau`
2. Créer le tableau des 100 premiers carrés par compréhension ainsi
   `carres = [ i**2 for i in range(1,101)]`.
   Que vaut difference(difference(carres)) ?
3. Observe -t-on une propriété similaire pour les cubes ? Généraliser.

Ex 13

Problème : Etant donné une chaîne de caractères comme 'Esope reste ici et se repose' considérée dans cet exercice comme un palindrome, écrire une fonction

`est_palindrome(chaine:str)->bool` qui prend en paramètre une chaîne de caractères et qui renvoie Vrai si cette chaîne est un palindrome et Faux sinon

Indication : Chercher des méthodes de la classe `str` qui :

1. renvoie une nouvelle chaîne en minuscules
2. renvoie une nouvelle chaîne sans espace

Ex 14

1. Peut on vous demander d'écrire une fonction Python qui change l'ordre des caractères dans une chaîne de caractères sur place ? Pourquoi ?
2. Ecrire une fonction `renverser(chaine)` qui prend en paramètre une chaine de caractères et qui renvoie une nouvelle chaine dont les caractères sont dans un ordre inverse à celui de la chaine passée en paramètre

Ex 15

1. On définit une fonction `f(n)` qui prend en argument un entier naturel `n` strictement positif de la manière suivante :
   (a) Si n est pair alors f(n) est égale à la moitié de n.
   (b) Sinon f(n) vaut trois fois n plus 1.

   Que vaut f(16) ? Que vaut f(5) ?
2. Définir `f(n)` dans le langage Python.
3. On observe que f(8) = 4 et f(4) = 2 et f(2) = 1.
   On matérialise cette suite de calculs par $8 \to 4 \to 2 \to 1$ et on dit que cette suite est de longueur 3. (On a appliqué trois fois la fonction f)
   Quelle est la suite de calculs lorsqu'on commence par 3 ? Quelle est la longueur de cette suite ?

4. On admet que pour tout entier naturel n strictement positif la suite de calculs précédents commençant par n arrive toujours en 1.

   n → ... →.....→ 1

   Par exemple 2022 → 1011 → 3034 →1517 .....→ 1 a pour longueur 63

   Ecrire une fonction en Python `longueur(n)` qui renvoie la longueur de la suite de calculs commençant par n.

5. On définit la hauteur d'une suite de calculs commençant par n et se terminant par 1 comme étant la plus grande valeur obtenue au cours des calculs.

   Par exemple

   la suite 3 → 10 → 5 → 16 → ...→ 1 a pour hauteur 16

   Ecrire une fonction en Python `hauteur(n)` qui renvoie la hauteur de la suite de calculs commençant par n.

6. Quel est le plus petit entier strictement plus petit que 1000 ayant la plus grande hauteur ?

   Pour résoudre ce problème proposer deux méthodes :

   (a) Une méthode où on recalcule les hauteurs sans mémoriser les calculs précédents

   (b) Une autre méthode où on mémorise les hauteurs calculées dans un tableau `hauteurs`

7. Quel est le plus petit entier strictement plus petit que 1000 ayant la plus grande longueur de suite de calculs ?