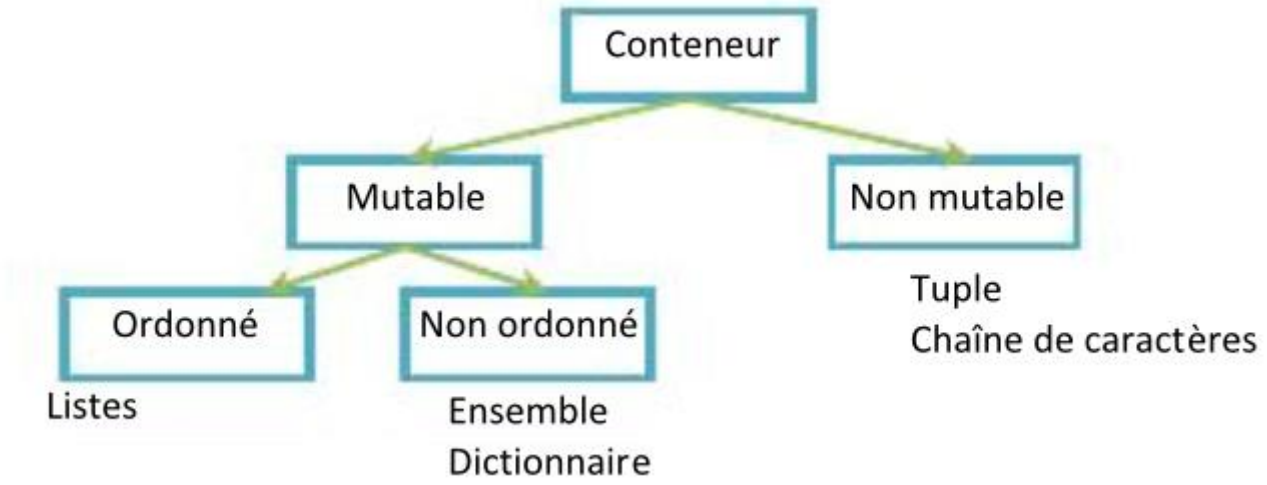
STRUCTURES DE DONNEES A 1 DIMENSION (2) : CORRIGE LES TUPLES



Me prévenir de toute erreur éventuelle.

- *Sites internet et logiciels* : pythontutor.com, éditeur et console Python (Thonny, VS Code etc.), franceioi.org.
- *Pré-requis pour prendre un bon départ :*

	☹	😐	☺	☺☺
Types de base.				
Listes.				
Représentation mémoire d'une variable par un couple lié nom-objet.				

Lorsque le logo Python apparait, cela signifie que l'activité doit être aussi faite sur ordinateur.

VERIFIEZ VOS REPONSES SUR ORDINATEUR !

NOM et prénom : …… *Première spécialité NSI*

Poursuivons notre fascinant voyage dans le monde des types de données en Python. Nous avons déjà vu :

- lors du cours 1 les types basiques (ou types natifs) principaux : booléens (bool), les types numériques entier (*int*) et non entier (représentation à virgule flottante : *float*).
- lors du cours 4 un premier type construit : le type séquentiel liste (*list*).
- le type caractère (*str*) qui est à cheval entre les types basiques et le type liste.

Ces différents types peuvent être différenciés selon un nouveau critère important : être modifiable ou pas.

I. OBJETS IMMUABLES ; OBJETS MUABLES.

On rappelle qu'en Python, les données, les valeurs de n'importe quel type font partie des objets.

A. Définition :

- Un objet **non modifiable** est dit **immuable** (immutable en anglais).
  Exemples : Les entiers, les floats, les booléens et les chaînes de caractères sont immuables.
- Inversement, un objet **modifiable** est dit **muable** (mutable en anglais).
  Exemple : Les listes font partie des objets modifiables.

Nous allons voir 2 nouveaux types construits : les tuples (ce livret) qui sont non modifiables et les dictionnaires (prochain livret) qui sont modifiables.

Compléter :	Types basiques	Types construits
Immuables	*int , float , bool*	*str , tuple*
Muables	∅	*list , dict*

B. Immuable ne veut pas dire non assignable !

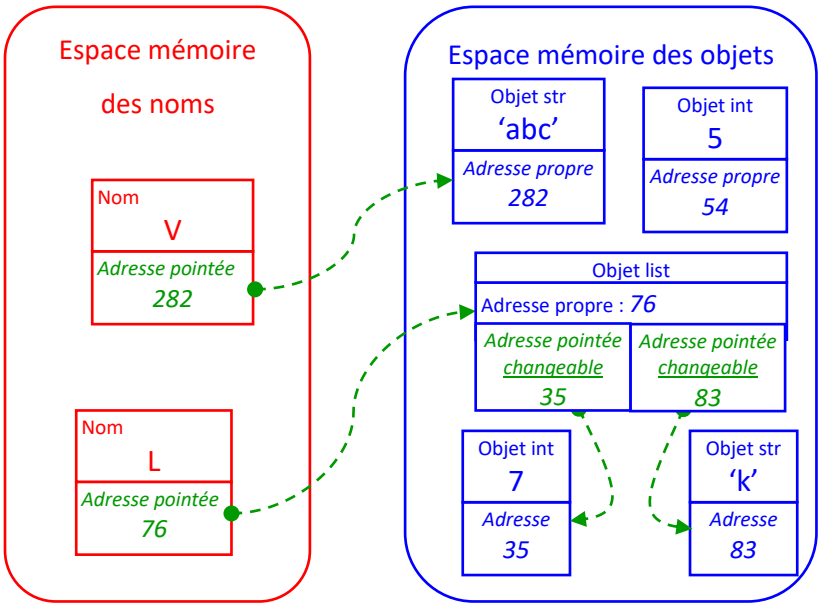
Un objet immuable est donc un objet constant. Cela ne veut pas dire qu'il ne peut pas changer de nom ! En effet, ne pas confondre nom et objet : le nom c'est l'identificateur, l'objet c'est la valeur.

Reprenons l'exemple de l'instruction V = 'abc' et sa représentation en mémoire (livret sur les listes p.10).

• 'abc' est affecté à V signifie que le nom V pointe vers l'objet 'abc'. Dire que la variable change par réaffectation (V = 5 par exemple) signifie que l'adresse pointée par le nom V changera (54 à la place de 282).

• Par contre l'objet 5 de type int lui ne peut pas changer : à l'adresse 54, il y aura toujours 5. De même, à l'adresse 282, il y aura toujours 'abc' et rien d'autre.

Par exemple l'instruction V.append('d') renverra AttributeError : on ne peut pas ajouter une lettre à une chaîne de caractère déjà formée.



• L'objet list en 76 est lui muable : les adresses pointées qui correspondent aux items peuvent changer.

Nous allons voir un objet immuable qui ressemble à une liste, mais dont les adresses pointées par ses items sont cette fois-ci fixes et ne peuvent plus être modifiées : *les tuples.*

II. TUPLES OU N-UPPLETS.

Puisqu'un tuple est une liste constante, on va retrouver dans ce qui suit beaucoup des caractéristiques des listes. Il faudra juste tenir compte du fait qu'un tuple n'est pas modifiable.

A. Débuter avec les tuples :

Définition	• En Python, un **tuple** est une ***liste non modifiable.*** Un tuple est donc un ensemble fini ordonné non modifiable d'objets indexés par des entiers. • **Les tuples, contrairement aux listes, font donc partie des objets immuables.**
Vocabulaire	• Les objets contenus dans un tuple s'appellent aussi les **éléments** de ce tuple. Les éléments ne sont pas forcément du même type ! Un tuple peut lui-même contenir d'autres tuples ou des listes etc. • Le numéro attaché à chaque objet d'un tuple s'appelle aussi **l'indice** de l'objet. Attention les indices d'un tuple commencent aussi à *0* et non à *1* !
Définir un tuple.	• Définir un tuple, c'est écrire tous les éléments le composant selon 2 règles : **entre parenthèses ( )** **séparés par des virgules.** • Exemples : • ( ) : tuple *vide*. Ne sert strictement à rien car on ne peut pas *modifier* un tuple ! • (1**,**) : tuple à 1 élément de type int. **Ne pas oublier la virgule !** Les tuples à un seul élément ne servent quasiment jamais sauf à générer des tuples d'éléments tous identiques ! (voir p.5) ⚠ un élément entre parenthèses sans virgule n'est pas un tuple mais juste l'élément ! Les parenthèses ne servent alors à rien. Ex : (1) est en fait juste 1. ('a') est en fait juste 'a'. • ('6ème1', '6ème2', '6ème3') : tuple de *3* éléments tous de type *str*. • (True , [1 , 3] , 2/3) : tuple de *3* items, 1 de type *bool*, 1 de type *list*, 1 de type *float*. • Remarque : Par analogie avec les Maths, un tuple à 2 éléments pourrait s'appeler un couple, un tuple à 3 éléments : un triplet, un tuple à 4 éléments : un quadruplet, etc., un tuple à n éléments : un *n-uplet*. Le mot « tuple » vient de la contraction de « table uplet ».
Nommer un tuple.	• C'est affecter le tuple à un nom de variable. Ex : point1 = (x1 , y1 , z1) point2 = (x2 , y2 , z2) points = (point1 , point2)
Type d'un tuple.	• Un tuple est un objet de type **tuple**. Ex : type((1 , 2)) renverra < class *'tuple'* >. « tuple » est donc un mot réservé de Python. Evitez de l'utiliser comme nom de qq chose !

B. Déjà une première bizarrerie ?!

Je ne l'ai pas dit mais un tuple, objet immuable, peut contenir des choses qui changent de valeur comme des variables ou des objets muables comme des listes !

Comment se comporte alors le tuple dans chacun de ces 2 cas ? Tester les scripts suivants :

Cas ① Tuple c contenant des variables :		Cas ② Tuple c contenant un objet muable (la liste b) :	
a , b = 1 , 2 c = (a , b) b = 3	Que vaut c ? Est-ce logique ? ……	a , b = 1 , [2] c = (a , b) b[0] = 3	Que vaut c ? Est-ce logique ? ……

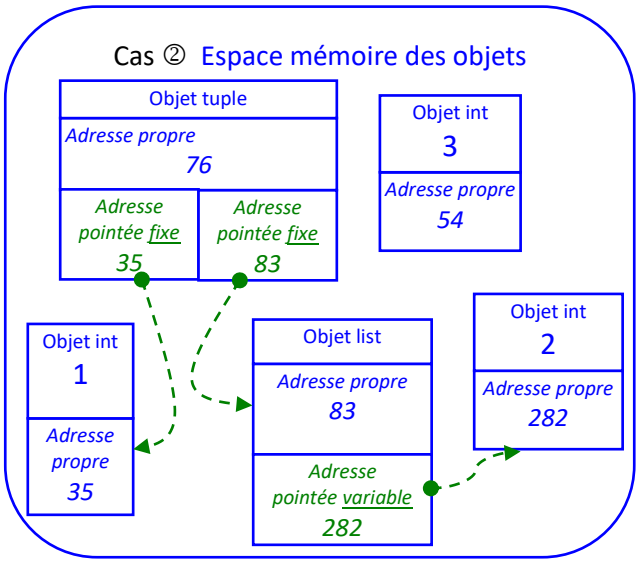
On nous avait vendu le type tuple comme quelque chose de non modifiable ! Nous aurait-on menti ?! 😑

Et bien non ! Là encore il faut regarder ce qui se passe en mémoire !

• Cas ① : quand la variable c est initialisée par le tuple (a , b), a est remplacé par l'adresse de l'objet int 1 et b est remplacé par l'adresse de l'objet int 2. Ces adresses pointées ne changent plus, à contrario des listes ! Même si b pointe après vers un autre objet int 3, le 2ème élément du tuple c pointe toujours vers l'objet int 2. Le tuple est bien resté constant !

• Cas ② : quand la variable c est initialisée avec le tuple (a , b), a est remplacé par l'adresse de l'objet int 1 et b est remplacé par l'adresse de l'objet list [ ] qui lui-même pointe vers l'objet int 2. Les adresses pointées dans le tuple ne peuvent plus changer.

Bien que le 2ème élément du tuple pointera toujours vers le même objet list [ ], cet objet list peut pointer vers un autre objet que 2 (l'objet 3 par exemple) et donc changer. Le tuple est bien resté constant, non au sens des valeurs, mais au sens des adresses pointées.



Un tuple est donc une liste non modifiable : après avoir été initialisées, les adresses des objets pointés par les éléments du tuple ne sont plus modifiables.

Une variable dans un tuple n'est évaluée qu'une seule fois puis sa valeur ne pourra plus changer (contrairement aux listes). Par contre, tout comme pour les listes, les objets muables (listes par exemple) dans un tuple peuvent voir leurs valeurs changer !

Que vaut le tuple c dans chacun des 4 scripts suivants :

a , b = 1 , 2 c = (a , b) a = 3 *c → (1 , 2)*	a , b = [1] , 2 c = (a , b) a[0] = 3 *c → ([3] , 2)*	a , b = [1] , 2 c = (a , b) b = 3 *c → ([1] , 2)*	a , b = 1 , [3] c = (a , b) a = b[0] *c → (1 , [3])*

⚠ un objet muable (liste entre autres) dans un tuple peut faire « varier » ce tuple !

C. Travailler avec les tuples ; comparaison avec les listes :

Voyons donc ce qu'on peut faire (ou pas) avec les tuples, en comparaison des listes.

En s'inspirant de la syntaxe pour les listes (à droite), compléter la colonne de gauche pour les tuples :

- en remplaçant le mot liste par le mot *tuple.*
- en remplaçant les crochets [ ] par des *parenthèses ( ).*

Création	
Création simple et initialisation d'un tuple : **nom_tuple = (1 , 'a')**	*Création simple et initialisation d'une liste :* **nom_liste = [1 , 'a']**
Créer un tuple de n éléments tous identiques : **nom_tuple = (élément,) * n ou n * (élément,)** **⚠ ne pas oublier la virgule !**	*Créer une liste de n éléments tous identiques :* **nom_liste = [élément] * n ou n * [élément]**
Créer le tuple d'une suite arithmétique d'entiers : **nom_tuple = tuple(range(d , f , p))**	*Créer la liste d'une suite arithmétique d'entiers :* **nom_liste = list(range(d , f , p))**
Compréhension de tuple ou tuple image : **(fonction(x) for x in séquence if condition)**	*Compréhension de liste ou liste image :* **[fonction(x) for x in séquence if condition]**
Créer une nouveau tuple3 par concaténation (accolement par l'opérateur +) de tuple1 et tuple2 : **tuple3 = tuple1 + tuple2**	*Créer une nouvelle liste3 par concaténation (accolement par l'opérateur +) de liste1 et liste2 :* **liste3 = liste1 + liste2**
Créer un tuple b en convertissant un objet itérable (chaîne str, range( ), liste, dictionnaire, etc.) : **b = tuple(objet itérable)**	*Créer une liste b en convertissant un objet itérable (chaîne str, range( ), tuple, dictionnaire, etc.) :* **b = list(objet itérable)**
Créer une tranche (slice) d'un tuple en allant de l'indice d inclus à l'indice f exclu. Et affecter cette tranche à b : **b = tuple[d : f]** ⚠ crochets et non parenthèses !	*Créer une tranche (slice) d'une liste en allant de l'indice d inclus à l'indice f exclu. Et affecter cette tranche à b :* **b = liste[d : f]**
Joindre des caractères à l'aide de la méthode .join( )	
Créer une nouvelle chaîne de caractères b à partir d'un tuple de plusieurs caractères : **b = "caractères de jointure".join(tuple)**	*Créer une nouvelle chaîne de caractères b à partir d'une liste de plusieurs caractères :* **b = "caractères de jointure".join(liste)**
Séparer les caractères d'une chaîne de caractères à l'aide de la méthode .split( )	
Créer un nouveau tuple a de caractères à partir d'une chaîne de caractères : **a = tuple("chaîne".split("caractères_coupure"))**	*Créer une nouvelle liste a de caractères à partir d'une chaîne de caractères :* **a = list("chaîne".split("caractères_coupure"))**

Tirer de l'information	
Nombre d'éléments d'un tuple (longueur du tuple) : **len(tuple)**	*Nombre d'éléments d'une liste (longueur de liste) :* **len(liste)**
Récupérer dans a l'élément d'indice k d'un tuple T : **a = T[k]** (crochets et non *parenthèses* !)	*Récupérer dans a l'élément d'indice k d'une liste L :* **a = L[k]**
Parcours d'un tuple T à l'aide d'une boucle for : • Parcours séquentiel à l'aide des indices : **for index in range(longueur de T) :** **traitement sur T[index]** • Parcours direct sur les éléments quand on n'a pas besoin des indices : **for élément in T :** **traitement utilisant élément**	*Parcours d'une liste L à l'aide d'une boucle for :* • Parcours séquentiel à l'aide des indices : **for index in range(longueur de L) :** **traitement sur L[index]** • Parcours direct sur les éléments quand on n'a pas besoin des indices : **for élément in L :** **traitement utilisant élément**
Récupérer le nb de fois où objet est dans tuple : **a = tuple.count(objet)**	*Récupérer le nb de fois où objet apparait dans liste :* **a = liste.count(objet)**
Oui ou non objet est-il présent dans tuple ? **objet in tuple**	*Oui ou non objet est-il présent dans liste ?* **objet in liste**
Récupérer le 1er indice d'objet dans tuple : **a = tuple.index(objet)**	*Récupérer le 1er indice d'objet dans liste :* **a = liste.index(objet)**
Copie	
• **Puisqu'un tuple est non modifiable, les copies de tuples sont forcément liées :** **tuple2 = tuple1** • Pas de méthode .copy( ) pour les tuples !	• *Copies liées de listes :* **liste2 = liste1** • *Copies non liées de listes :* **liste2 = liste1.copy( )**

D. Ce qu'on ne peut logiquement pas faire avec un tuple !

Rappelons qu'un tuple n'est pas modifiable : les adresses pointées par ses éléments ne sont pas modifiables.

Donc toutes les méthodes et fonctions assurant la maintenance d'une liste et qui donc la modifient sur place (c-à-d qui changent la liste elle-même) donnent Error pour les tuples :

- **Impossible dans un tuple de modifier un élément ou de modifier la place d'un élément :**

  Donc tuple[k] = nouvellevaleur, reversed(tuple) et tuple.reverse( ), sorted(tuple) et tuple.sort( ) → Error.

- **Impossible d'enlever ou d'ajouter des éléments à un tuple :**

  Donc tuple.clear( ), del tuple[k], tuple.remove( ), tuple.append( ), tuple.insert( ), tuple.extend( ) → *Error*

E. Pourquoi les tuples sont-ils quand même si importants ?

Faut-il encore le rappeler, un tuple est comme une liste *immuable.*

On pourrait alors légitimement se dire que les listes suffisent et que si l'on a besoin d'une liste constante, il suffit de ne pas y toucher n'est-ce pas ? Et bien non, les tuples ont au moins 5 bonnes raisons d'exister !

Les tuples permettent la transmission sécurisée d'informations !

C'est la principale raison d'être des tuples !

Grâce aux tuples, on peut être certain que des informations sensibles (par exemple des couples login-mot de passe) ne seront pas modifiées par la suite des traitements !

Ainsi beaucoup d'informations sont souvent structurées sous forme de tuples. En voici quelques exemples :

- Les coordonnées d'un lieu : (latitude , longitude).
- L'identité d'une personne : (NOM , Prénom , n° d'identité).
- La couleur des points d'une image mise sous la forme d'un triplet (R , V , B).

Un tuple est moins lourd qu'une liste.

Un tuple prend moins de place en mémoire qu'une liste ayant les mêmes éléments.

Voyons cela avec la fonction getsizeof (du module sys : system) qui renvoie le poids en octets d'un objet :

```
from sys import getsizeof
```

getsizeof((1 , 2)) → *36 octets.*

getsizeof([1 , 2]) → *44 octets.*

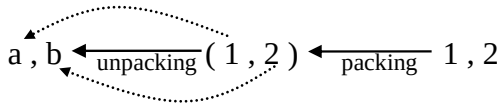
Qu'en déduit-on ? *poids(tuple) < poids(liste)*

Les tuples permettent les affectations parallèles.

Rappel affectation parallèle sur un exemple : a , b = 1 , 2 ⇔ b = 2 puis a = 1.

• En fait 1 , 2 est d'abord transformé en le tuple (1 , 2) : c'est l'emballage en tuple de valeurs **(packing).**

• Puis chaque élément du tuple est dispatché parallèlement sur chaque variable b et a : c'est le déballage du tuple (**unpacking**).



• Lors du déballage, grâce à la notation *variable, on n'est même pas obligé d'avoir le même nombre de variables réceptrices que d'éléments à dispatcher dans le tuple.

Exemple : début, *f , b , fin = 1 , 2 , 3 , 4 , 5 , 6

début → 1 fin → 6 b → 5 donc f → [2 , 3 , 4]

f a ramassé le reste des éléments sous forme de liste.

a , b , c , *d , e = 1 , 2 , 3 , 4 , 5 , 6 , 7

a → 1 b → 2 c → 3 e → 7

donc d → [4 , 5 , 6]

Remarques :

• Le déballage marche aussi pour les listes : soit a , *b = [0 , 1 , 2] alors *a → 0 donc b → [1 , 2].*

• L'affectation parallèle permet de facilement permuter 2 variables sans variable auxiliaire : **b , a = a , b**

• L'affectation parallèle est fondamentale pour les fonctions qui retournent plusieurs valeurs.

Les fonctions renvoient leurs résultats multiples sous forme d'un tuple.

Lorsqu'une fonction renvoie plus qu'une valeur, ces valeurs sont retournées sous forme d'un tuple.

Ce tuple renvoyé peut être déballé dispatché sur plusieurs variables par affectation parallèle.

Exemple :

```
def essai( ) :
    return 1 , 2 , 3 , 4
a , *b , c = essai( )
```

a → 1 c → 4 donc b → [2 , 3]

```
def minmax(L) :
    Locale = L[ : ]
    return min(Locale), max(Locale)
a = minmax ([ 2 , 5 , 7 , 1 ])
```

a → (1 , 7).

Remarque : Le fait que les fonctions renvoient les résultats multiples sous forme d'un tuple et non d'une liste est une sécurité surtout lorsque plusieurs processus fonctionnent en même temps (multithreading) en partageant les mêmes ressources (fonctions, modules etc.). Le tuple retourné par une fonction utilisée dans un processus ne pourra être modifié lors de l'utilisation de la même fonction par un autre processus.

Les tuples jouent peuvent jouer le rôle de clef dans les dictionnaires :

Contrairement aux listes.

Nous verrons cela dans le livret suivant sur les dictionnaires.

Ai-je tout compris ? Tuples.	☹	😐	☺	☺☺
Objets immuables, muables.				
Définition et vocabulaire des tuples.				
Tuple contenant une liste : que se passe-t-il ?				
Méthodes et fonctions communes aux tuples et aux listes.				
Méthodes et fonctions qui marchent pour les listes mais pas pour les tuples.				
Intérêt des tuples.				
Poids d'un tuple par rapport à une liste.				
Affectation parallèle : packing et unpacking d'un tuple.				
Renvoie de valeurs multiples par une fonction.				
Tuple et dictionnaire.				