4

Tuple et Tableau

Extrait du programme

Thème : Types construits

Contenus : p-uplets.

Capacités attendus : Écrire une fonction renvoyant un p-uplet de valeurs.

Commentaires :

Un tableau de Melinda Copper

Thème : Types construits

Contenus :

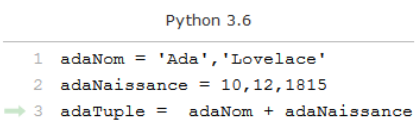
Tableau indexé,
tableau donné en compréhension

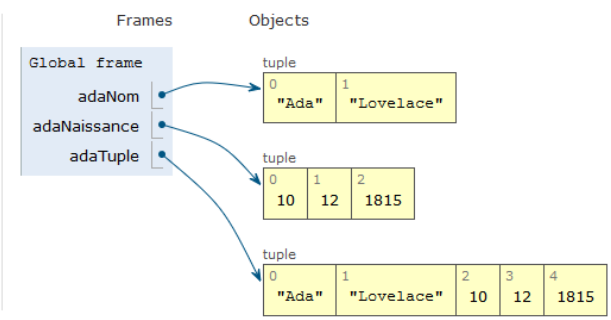
Capacités attendus :

Lire et modifier les éléments d'un tableau grâce à leurs index.
Construire un tableau par compréhension.
Utiliser des tableaux de tableaux pour représenter des matrices : notation `a[i][j]`.
Itérer sur les éléments d'un tableau.

Commentaires :

Seuls les tableaux dont les éléments sont du même type sont présentés.
Aucune connaissance des tranches (slices) n'est exigible.
L'aspect dynamique des tableaux de Python n'est pas évoqué.
Python identifie listes et tableaux.
Il n'est pas fait référence aux tableaux de la bibliothèque NumPy.





I. Tuple

a) Définition

D'après Wikipédia : *En mathématiques, si n est un entier naturel, alors un n-uplet ou n-uple est une collection ordonnée de n objets.*
Par exemple les coordonnées d'un point $(5;1)$ est 2-uplet, une date de naissance (31;décembre;2000) est un 3-uplet.
En informatique, on utilise le terme anglais **tuple**. D'après Wikipédia [1], il s'agirait de l'abréviation de quin**tuple**, sex**tuple**.
D'après Wiktionnaire [2], ce serait le néologisme **t**able **uple**t, une table de n-uplet.
Peut-être ne le saurons-nous jamais ...

Voici une définition plus précise de la notion de tuple dans un cadre informatique :

Définition :
Un **tuple** est un objet composé de différents éléments ordonnés, avec un indice, **non modifiable**s.

b) Création d'un tuple

- Pour créer un tuple il suffit d'écrire des valeurs séparées par des virgules. Ces valeurs ne sont pas obligatoirement du même type.

```
>>> adaTuple = 'Ada','Lovelace',10,12,1815
>>> adaTuple
('Ada', 'Lovelace', 10, 12, 1815)
```

- On peut aussi mettre les valeurs entre parenthèses (D'une manière générale, on préfèrera cette notation).

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> adaTuple
('Ada', 'Lovelace', 10, 12, 1815)
```

- Un tuple peut contenir un autre tuple...

```
>>> alanTuple = 'Alan','Turing',(23,6,1912)
>>> alanTuple
('Alan', 'Turing', (23, 6, 1912))
```

- Si le tuple est composé d'une seule valeur il faut l'écrire `t=3,` ou `t=(3,)`. Attention la virgule est obligatoire. (A nouveau on priviligiera la notation avec les parenthèses)

```
>>> t=3,
>>> t
(3,)
```

- Pour tuple vide `t=()`.

```
>>> t=()
>>> t
()
```

En résumé :
Les virgules sont toujours obligatoires, pas les parenthèses... sauf pour tuple vide.

1. https://fr.wikipedia.org/wiki/N-uplet
2. https://fr.wiktionary.org/wiki/tuple

c) Accès aux éléments

Si un tuple est non vide, ses n éléments sont indexés par les entiers 0,1,2,...,$n-1$ comme pour les chaînes de caractères.

- On accède à chacun de ses éléments en utilisant des crochets et l'indice en élément.

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> adaTuple[0]
'Ada'
>>> adaTuple[2]
10
>>> alanTuple = ('Alan','Turing',(23,6,1912))
>>> alanTuple[2]
(23, 6, 1912)
>>> alanTuple[2][0]
23
```

- Un dépassement de l'indexe $n-1$ renvoie une erreur.

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> adaTuple[5]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: tuple index out of range
```

- La longueur d'un tuple est donnée par la fonction `len()`.

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> len(adaTuple)
5
>>> alanTuple = ('Alan','Turing',(23,6,1912))
>>> len(alanTuple)
3
```

- On peut accéder aux éléments en partant de la fin du tuple, l'indice `-1` étant une abréviation de `len(t)-1`.

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> adaTuple[-1]
1815
>>> adaTuple[-3]
10
```

- Les éléments d'un tuple ne sont pas modifiable.

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> adaTuple[1] = 'comtesse de Lovelace'
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

d) Affectation multiple

- On peut affecter à plusieurs variables en même temps à l'aide d'un tuple.

```
>>> a,b,c = 4,5,6
>>> a
4
```

- On peut alors échanger simplement les valeurs de plusieurs variables. (ce qui n'est pas le cas dans d'autres langages)

```
>>> a,b = 4,5
>>> a,b = b,a
>>> a
5
>>> b
4
```

- On peut enfin affecter à plusieurs variables les éléments d'un tuple. Il faut juste s'assurer d'avoir autant de variables que d'éléments dans le tuple.

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> prenom,nom,jour,mois,annee = adaTuple
>>> nom
'Lovelace'
>>> jour
10
```

e) Opérations

- Le symbole + effectue une concaténation de deux tuples (comme avec les chaînes de caractères).

```
>>> adaNom = ('Ada','Lovelace')
>>> adaNaissance = (10,12,1815)
>>> adaTuple =  adaNom + adaNaissance
>>> adaTuple
('Ada', 'Lovelace', 10, 12, 1815)
```

- Le symbole * répète plusieurs fois la concaténation du même tuple.

```
>>> tuple = ('je','me','répète')
>>> 3*tuple
('je', 'me', 'répète', 'je', 'me', 'répète', 'je', 'me', 'répète')
```

f) Fonction renvoyant un tuple

Division euclidienne

Nous avons déjà vu au chapitre 3, **Représentation des entiers**, que nous avions besoin de la division euclidienne, c'est-à-dire de la valeur du quotient entier ET du reste.
Nous pouvons créer une fonction qui va nous renvoyer ces 2 valeurs sous la forme d'un tuple.
Ligne 4, les parenthèses ne sont pas obligatoires mais il s'agit du tuple `(quotient, reste)`.

```
1 def divEuclienne(a,b):
2       quotient = a//b
3       reste = a%b
4       return quotient, reste
```

On obtient les résultats suivants :

```
1 >>> divEuclidienne(10,2)
2 (5, 0)
3 >>> divEuclidienne(13,2)
4 (6, 1)
5 >>> reste = divEuclidienne(13,2)[1]
6 >>> reste
7 1
```

Une série de notes

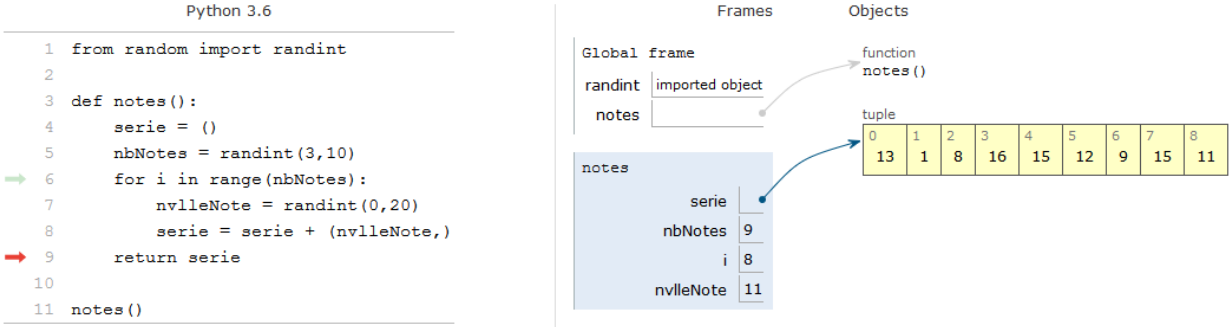
La fonction suivante permet de créer un tuple de longueur aléatoire entre 3 et 10, contenant des nombres entiers choisis aux hasards entre 0 et 20. Cette fonction va nous servir à simuler l'ensemble des notes sur un trimestre dans une matière.

```
1 from random import randint
2
3 def notes():
4     serie = ()
5     nbNotes = randint(3,10)
6     for i in range(nbNotes):
7         nvlleNote = randint(0,20)
8         serie = serie + (nvlleNote,)
9     return serie
```

Nous obtenons par exemple le résultat suivant :

```
1 >>> notes()
2 (13, 1, 8, 16, 15, 12, 9, 15, 11)
```

Nous pouvons regarder ce qui se passe, pas à pas, en ligne sur le site http://pythontutor.com/ :



- Mais Monsieur, votre truc, là, il arrête pas de modifier le tuple ! Sa longueur est de plus en plus grande !
- Oui tu as tout à fait raison, petit scarabée.
- Vous nous auriez menti ! ? ? ?
- Pas tout à fait.
- C'est-à-dire ?
- A chaque itération on crée un nouveau tuple qui va contenir l'ancien plus le tuple contenant la nouvelle note, puis on efface de la mémoire l'ancien qui porte le même nom.
- Vous voulez dire qu'à chaque étape, le tuple `serie` *n'est plus vraiment le même, bien qu'il porte le même nom ?*
- Oui, petit scarabée. Tu as compris.
- Hé hé hé !
- On peut d'ailleurs s'en rendre compte si on demande l'adresse mémoire du tuple `serie` *avant et après ajout de la nouvelle note. Dans le code ci-dessous, c'est aux lignes 10 et 12.*

```
1  from random import randint
2
3  def notes():
4      serie = ()
5      nbNotes = randint(3,10)
6      for i in range(nbNotes):
7          nvlleNote = randint(0,20)
8          print("avant",id(serie))
9          serie = serie + (nvlleNote,)
10         print("apres",id(serie))
11     return serie
```

La console donne :

```
1 >>> notes()
2 avant 895051038792
3 après 895125396728
4 avant 895125396728
5 après 895195801544
6 avant 895195801544
7 après 895125264496
8 (10, 7, 5)
```

- On constate bien que l'adresse mémoire du tuple `serie` *n'est pas la même avant et après modification. Ce n'est donc pas le même tuple.*
- Hé ! Hé ! Hé ! J'avais bon !
- Oui et maintenant c'est à toi de coder petit scarabée.
- Ups !

g) Quelques exercices sur les tuples

Exercice 1

1. Créer une fonction qui prend en argument un tuple de notes et qui renvoie un tuple donnant, dans cet ordre la plus petite note, la moyenne et la plus grande note.
   Entrée : Un tuple de n notes.
   Sortie : Un tuple donnant dans l'ordre la plus petite note, la moyenne des notes et la plus grande note.
   Exemples :
   `statNotes((13, 1, 8, 16, 15, 12, 9, 15, 11))` doit renvoyer `(1, 11.11, 16)`.
   `statNotes((4, 2, 3, 0, 6, 15, 11, 16, 6, 13))` doit renvoyer `(0, 7.6, 16)`.
   Aide : On peut commencer par faire une fonction qui calcule la moyenne, puis une autre qui recherche la plus petite note, enfin la note la plus grande.
2. Tester votre fonction avec la fonction `notes()` donnée précédemment.
3. Créer un programme qui crée 3 listes de notes en NSI, en Français et en Histoire-Géographie.
   En utilisant les coefficients suivants, 10 pour la NSI, 5 pour le Français et 3 en Histoire-Géographie, votre programme doit vous dire si vous aurez votre BAC ... ou pas.

Exercice 2 (Triplet Pythagoricien)
Un professeur de mathématiques fainéant cherche des triplets Pythagoriciens[3] différents du classique $(3, 4, 5)$. Il demande donc à ses élèves de NSI de lui écrire un programme qui en trouve aléatoirement un pour des longueurs entières comprises entre 1 et 100. Seule demande particulière, la dernière longueur doit être la plus grande (et donc être la longueur de l'hypoténuse).
- Mais Monsieur, je le connais ce prof !
- Je ne pense pas petit scarabée.

Exercice 3 (Triplet Pythagoricien **)
Et si on trouvait le nombre[4] de triplets Pythagoriciens composés de nombres entiers inférieurs à 100 ? ? ?

Exercice 4 (Triplet Pythagoricien *)**
Et si on estimait le nombre d'essais nécessaire à notre premier programme pour trouver un triplet Pythagoricien composés de nombres entiers inférieurs à 100 ? ? ?

h) Limites

- Mais Monsieur ! on ne peut pas modifier la longueur d'un tuple ni les différentes valeurs qui la composent. C'est un peu limité votre truc !
- Oui, petit scarabée. Un tuple est une collection de taille constante de valeurs constantes.
- Ben c'est ce que je dis !
- C'est quand même bien pratique de pouvoir stocker un ensemble de valeurs, même si nous ne pouvons pas les modifier. D'ailleurs Ada Lovelace n'a aucune raison de changer de nom ou de date de naissance :
`adaTuple = ('Ada','Lovelace',10,12,1815)`.
- ...
- Nous verrons dans la suite de cours la notion de tableau qui permettent de dépasser cette limite.
- Je suis (pas) pressé de voir ça... Mais Monsieur, j'y pense, votre truc est nul, parce que mon cousin il vit aux USA, et là-bas, ils inversent le jour et le mois. Avec votre tuple `adaTuple = ('Ada','Lovelace',10,12,1815)`*, en France, on pense qu'Ada est née le 10 décembre 1815, alors qu'aux states, on pense qu'elle est née le 12 octobre 1815... Hé hé hé !*
- C'est bien, tu penses.
- ...
- Et, oui, avec un tuple, l'ordre dans lequel les valeurs sont données a son importance. Il faut donc se donner des règles bien strictes sur la définition de notre tuple. Par exemple, pour le point de coordonnées $(2, 5)$*, on sait que l'abscisse est 2 et l'ordonnée est 5. C'est une règle arbitraire que nous avons choisie. Mais dans d'autres cas, c'est moins clair comme tu viens de le remarquer petit scarabée.*
- Hé hé !
- Nous verrons plus tard la notion de dictionnaire qui permet d'associer une clef à chaque valeurs.
- Oh non ! Pitié !

3. Un triplet Pythagoricien est un triplet de trois nombres correspondant aux longueurs des 3 côtés d'un triangle rectangle. Par exemple $(3, 4, 5)$ en est car $3^2 + 4^2 = 5^2$.

4. Il y a exactement 16 triplets primitifs : (3, 4, 5) (20, 21, 29) (11, 60, 61) (13, 84, 85) (5, 12, 13) (12, 35, 37) (16, 63, 65) (36, 77, 85) (8, 15, 17) (9, 40, 41) (33, 56, 65) (39, 80, 89) (7, 24, 25) (28, 45, 53) (48, 55, 73) (65, 72, 97) d'après mon ami Wikipédia.
`https://fr.wikipedia.org/wiki/Triplet_pythagoricien`.
Les autres triplets sont obtenus par proportionnalité. Ainsi en multipliant par 2 (3, 4, 5), on a (6, 8, 10), en multipliant par 3, on a (9, 12, 15), ...

II. Hors programme mais parfois bien pratique : le slicing

Le **slicing** est une technique particulière qui permet de récupérer une série d'éléments d'un tuple à partir d'une borne, jusqu'à une borne, ou entre deux bornes.

- Donner tous éléments à partir de celui indexé 2 (inclus) : `tuple[2:]`

```
>>> adaTuple = ('Ada','Lovelace',10,12,1815)
>>> adaTuple[2:]
(10, 12, 1815)
```

- Donner tous éléments jusqu'à celui indexé 2 (exclu) : `tuple[:2]`

```
>>> adaTuple[:2]
('Ada', 'Lovelace')
```

- Donner tous éléments entre celui d'index 1 (inclus) et celui d'index 3 (exclu) : `tuple[1:3]`

```
>>> adaTuple[1:3]
('Lovelace', 10)
```

III. Tableau

a) Définition

Mon ami wikipédia[5] me donne la définition suivante pour un tableau :
« *En informatique, un* ***tableau*** *est une structure de données représentant une séquence finie d'éléments auxquels on peut accéder efficacement par leur position, ou indice, dans la séquence. C'est un type de conteneur que l'on retrouve dans un grand nombre de langages de programmation.* »

...

Malheureusement cette structure n'est pas implémenté en Python.
Cool. Le chapitre est terminé.
Au revoir et à la semaine prochaine.

5. `https://fr.wikipedia.org/wiki/Tableau_(structure_de_données)`

III. Tableau

a) Définition

Bon, sérieusement, un tableau c'est quoi ?

Définition :
Un **tableau** est un ensemble ordonné, de taille fixe, d'éléments d'un même type.
On peut accéder à chaque élément d'un tableau, récupérer sa valeur et/ou la modifier.

Avec Python, nous utiliserons le type `list` qui est beaucoup plus permissif. Voyons plutôt.

b) Ensemble ordonné d'éléments de même type

En Python, une liste est un ensemble ordonné d'éléments *qui ne sont pas nécessairement d'un même type*. Ces éléments sont écrits entre crochets séparés par des virgules.

- Ce que l'on peut faire en Python, **mais** qu'il ne faut pas faire :

```
>>> T = [12, "azerty", 3.1415927, ("Alan", "Turing")]
```

`T` est une *liste* ordonnée de ..... éléments :
.....
.....
Bien ce soit possible en Python, cette écriture est déconseillée, hors programme et interdite au lycée [6].

- Ce qu'il faut faire :

```
>>> T = [3, 1, 2020]
>>> U = ["Alan", "Turing"]
>>> V = [3.1415927, 1.414]
```

Les listes `T`, `U` et `V` contiennent des éléments ordonnés du même type. On considérera alors que ce sont des **tableaux**.
`T` est un tableau de .....
`U` est un tableau de .....
`V` est un tableau de .....

6. J'espère avoir été assez clair !

c) Les éléments sont modifiables

- On peut avoir accès à tout élément d'un tableau par son index. On peut alors récupérer cet élément et/ou le modifier. Attention ! Vous en avez maintenant l'habitude, on commence à compter à partir de 0 et si le tableau est de longueur n, le dernier élément est associé à l'index $n-1$.
  De plus, on peut compter à rebours, de la droite vers la gauche, en partant du dernier élément indexé -1.

```
>>> T = [3, 4, 8, 9, 10, 11, 45, 1, 6]
>>> T[0]
3
>>> T[1]
4
>>> T[9]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: list index out of range
>>> T[8]
6
>>> T[-1]
6
>>> T[-9]
3
```

- En ayant un accès à un élément d'un tableau on peut aussi le modifier.

```
>>> T = [3, 4, 8, 9, 10, 11, 45, 1, 6]
>>> T[0]=2020
>>> T
[2020, 4, 8, 9, 10, 11, 45, 1, 6]
```

d) Un tableau est de taille fixe

Attention, en Python, une liste **n'est pas de** taille fixe.

- **Hors - Programme :** En Python, on peut utiliser la méthode `append()` pour rajouter un élément à la fin de la liste et la méthode `pop()` pour enlever le dernier élément de la liste. Ces deux notions sont totalement hors-programme, mais parfois bien pratiques. Il est donc possible que parfois je les utilise. Bien évidemment, aucune connaissance sur ces 2 méthodes n'est exigibles :

```
>>> T = [3, 4, 8, 9, 10, 11, 45, 1, 6]
>>> T.append(2)
>>> T
[3, 4, 8, 9, 10, 11, 45, 1, 6, 2]
>>> T.pop()
2
>>> T
[3, 4, 8, 9, 10, 11, 45, 1, 6]
```

e) Quelques outils et méthodes intéressantes en Python

- On peut connaître la longueur d'une liste avec la fonction `len()` :

```
>>> T = [1,2,3]
>>> len(T)
3
```

- On peut concaténer plusieurs listes avec l'opérateur `+` :

```
>>> T = [1, 2, 3]
>>> U = [4, 5, 6]
>>> V = T + U
>>> V
[1, 2, 3, 4, 5, 6]
```

- On peut répéter plusieurs fois une liste avec l'opérateur `*` :

```
>>> T = [1, 2, 3]
>>> U = 3 * T
>>> U
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- On peut inverser l'ordre d'une liste avec la méthode `reverse()` :

```
>>> T = [1, 2, 10, 5, 8, 3]
>>> T.reverse()
>>> T
[3, 8, 5, 10, 2, 1]
```

- On peut trier une liste avec la méthode `sort()` :

```
>>> T = [1, 2, 10, 5, 8, 3]
>>> T.sort()
>>> T
[1, 2, 3, 5, 8, 10]
```

IV. Construction et parcours d'un tableau

Dans cette section nous allons créer, de 2 façons différentes, un tableau contenant les 100 premiers carrés parfaits, de 0^2 à 99^2.

a) Construction à l'aide d'un boucle pour

On commence par créer un tableau rempli de 0.

```
#On crée on conteneur de la bonne dimension sous la forme d'un tableau contenant 100 fois le nombre 0
t = [0]*100
#on remplit le tableau
for i in range(100):
    t[i] = i**2
```

b) Construction par compréhension

Cette technique permet d'écrire la boucle pour à l'intérieur des crochets de la liste. C'est plus court à écrire mais parfois plus compliqué à comprendre.
Cette définition prend la forme suivante :

```
[fonction(item) for item in iterable if condition(item)]
```

```
t = [i**2 for i in range(100)]
```

Et si par exemple, on veut tous les carrés parfaits pairs (donc le reste de la division par 2 est 0) :

```
t = [i**2 for i in range(100) if (i**2)%2==0]
```

c) Parcourir un tableau par son indice

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

d) Parcourir un tableau par ses éléments

```
for element in t:
    print(element)
```

V. Copie (Hors-Programme)

Attention, ceci est une partie très délicate du cours qui peut aboutir à de nombreuses erreurs...

Le type `list` est un objet qui permet de modifier les valeurs de chaque éléments de la liste. Malheureusement cette souplesse à un impact sur la copie de listes. Voyons plutôt :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2 #tout va bien
[1, 2, 3]
>>> liste1[1] = 5
>>> liste1
[1, 5, 3]
>>> liste2 #ben ça alors
[1, 5, 3]
```

En réalité, les deux noms `liste1` et `liste2` font référence à un seul et même objet. On peut le vérifier en utilisant la fonction `id()` qui nous donne l'identité de chaque objet, autrement dit l'adresse mémoire de l'objet :

```
>>> id(liste1)
676146267336
>>> id(liste2)
676146267336
```

On constate bien que les 2 noms font référence à la même adresse. C'est donc qu'il s'agit du même objet. Si on modifie `liste1`, on modifie bien `liste2`.

On peut aussi visualiser ce référencement à l'aide du site PythonTutor : http://pythontutor.com/

Pour obtenir deux listes différentes, il faut faire ce qu'on appelle une **copie profonde** à l'aide de la fonction `deepcopy()` de la bibliothèque `copy` :

```
>>> from copy import deepcopy
>>> liste1 = [1, 2, 3]
>>> liste2 = deepcopy(liste1)
>>> liste2 #tout va bien
[1, 2, 3]
>>> liste1[1] = 5
>>> liste1
[1, 5, 3]
>>> liste2 #je suis rassuré
[1, 2, 3]
>>> id(liste1)
676146263304
>>> id(liste2) #oui ce n'est pas le même adresse
676132758536
```

A nouveau un lien vers Pythontutor pour voir ce qui se passe : http://pythontutor.com/

VI. Exercices sur les tableaux

Exercice 5 (Multiples, Diviseurs *)

1. Écrire une fonction `multiples1(n)` qui prend en paramètre un entier naturel non nul `n` et renvoie un tableau contenant les 10 premiers multiples non nuls de `n`.
2. Écrire une fonction `multiples2(n)` qui prend en paramètre un entier naturel non nul `n` et renvoie un tableau contenant la liste des multiples de `n` inférieurs strictement à 1000.
3. Écrire une fonction `diviseurs(n)` qui prend en paramètre un entier naturel non nul `n` et renvoie un tableau contenant la liste des diviseurs de `n`.

Exercice 6 (Multiples, Diviseurs par compréhension **)

Reprendre l'exercice précédent réécrivant les fonctions à l'aide d'une construction par compréhension.

Exercice 7 (Multiplier un tableau *)

Écrire une fonction `produit(nombres,n)` qui prend en paramètres un tableau de nombres `nombres` et un entier naturel non nul `n` et renvoie un nouveau tableau obtenue en multipliant chaque élément du tableau `nombres` par `n`.

Exercice 8 (Manipulations de tableaux *)**

1. Écrire une fonction `tableau_aleatoire()` qui renvoie un tableau de 20 valeurs prises aléatoirement entre 1 et 10.
2. Écrire une fonction `occurrence(tab,n)` qui prend en argument un tableau `tab` de valeurs prises entre 1 et 10 et l'argument `n` un nombre entre 1 et 10. Cette fonction devra compter le nombre d'occurrence du nombre `n` dans le tableau `tab`.
3. Écrire une fonction `tableau_croissant(tab)` qui prend en argument un tableau `tab` de valeurs prises entre 1 et 10 et qui renvoie le tableau contenant tous les éléments du tableau `tab` sans aucune répétition, classé dans l'ordre.
4. Écrire une fonction `jointure(tab1, tab2)` qui prend en arguments deux tableaux `tab1` et `tab2` de valeurs prises entre 1 et 10. Cette fonction retourne un tableau contenant tous les éléments de `tab1` et `tab2` sans aucune répétition, classé dans l'ordre.

Exercice 9 (Manipulations de tableaux 2 *)**

1. Écrire une fonction `prefixe(tab1, tab2)` qui renvoie `True` si le tableau `tab1` est un préfixe du tableau `tab2`, autrement dit `tab2` commence exactement par les éléments de `tab1` dans le même ordre.
2. Écrire une fonction `suffixe(tab1, tab2)` qui renvoie `True` si le tableau `tab1` est un suffixe du tableau `tab2`, autrement dit `tab2` termine exactement par les éléments de `tab1` dans le même ordre.
3. Écrire une fonction `hamming(tab1, tab2)` qui renvoie le nombre d'indices auxquels les deux tableaux `tab1` et `tab2` sont différents.

VII. Table

Une table est un tableau de tableaux.
Pour être plus précis, on peut dire qu'une table contient plusieurs lignes et que chaque ligne est un tableau.
Donc, une table est ce qu'on appelle plus communément ... un tableau à 2 dimensions !
En mathématiques, on appelle cela une matrice.

Voici donc une table :

1	2	3	4
5	6	7	8
9	10	11	12

Cette table est composée de 3 tableaux : | 1 | 2 | 3 | 4 |, | 5 | 6 | 7 | 8 | et | 9 | 10 | 11 | 12 |.
On va pouvoir la coder en Python ainsi :

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

La commande `table[i]` nous permet alors d'avoir accès à la `i`-ème ligne.

```
>>> table[1]
[5, 6, 7, 8]
```

La commande `table[i][j]` nous permet d'avoir accès à la `j`-ème case de la `i`-ème ligne.

```
>>> table[1][2]
7
```

VIII. Exercices sur les tables

Exercice 10 (A faire à la main *)

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

1. Que retourne la commande `table[0][1]` ? .....
2. Quelle commande retourne `8` ? .....
3. Quelle commande retourne `1` ? .....
4. Quelle commande retourne `4` ? .....

Exercice 11 (Dimension d'un table *)
Écrire une fonction `dim()` qui prend en argument une table et qui retourne sa dimension sous la forme d'un tuple.
Par exemple `dim([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])` retourne le tuple `(3,4)`, pour 3 lignes et 4 colonnes.

Exercice 12 (Carré magique *)**
Un carré magique d'ordre n est une table composé de n lignes et de n colonnes, chaque case contenant un nombre strictement positif.
Les nombres sont disposés de sorte que :

- la somme des éléments d'une ligne est toujours égale à la constante magique S ;
- la somme des éléments d'une colonne est toujours égale à la constante magique S ;
- la somme des éléments d'une diagonale est toujours égale à la constante magique S.

De plus, un carré magique contenant $n \times n$ nombres est dit normal s'il contient tout les nombres de 1 à n^2.

Le tableau suivant est un carré magique normal.

4	9	2
3	5	7
8	1	6

En effet :

- La somme des éléments de chaque ligne vaut 15 :
  $4+9+2 = 3+5+7 = 8+1+6 = 15$
- La somme des éléments de chaque colonne vaut 15 :
  $4+3+8 = 9+5+1 = 2+7+6 = 15$
- La somme des éléments de chacune des deux diagonales vaut 15 :
  $4+5+6 = 2+5+8 = 15$

De plus ce carré magique est normal car il contient tout les nombres de 1 à $3^2 = 9$.

1. Écrire une fonction `somme_ligne(table,i)` qui calcule la somme de tous les nombres de la ligne `i`.
2. Écrire une fonction `somme_colonne(table,j)` qui calcule la somme de tous les nombres de la colonne `j`.
3. Écrire une fonction `somme_diagonale_hgbd(table)` qui calcule la somme de tous les nombres de la diagonale du coin en haut à gauche au coin en bas à droite.
4. Écrire une fonction `somme_diagonale_bghd(table)` qui calcule la somme de tous les nombres de la diagonale du coin en bas à gauche au coin en haut à droite.
5. Écrire un programme `carre_magique(table)` qui retourne `True` si la `table` est un carré magique.
6. Écrire un programme `normal(table)` qui retourne `True` si la `table` est un carré normal (tous les nombres de 1 à n^2 sont présents).
7. Écrire un programme `carre_magique_normal(table)` qui retourne un tuple contenant deux booléens : `True` si la `table` est un carré magique et `True` si la `table` est un carré normal.

8. Vérifier vos programmes avec les tables suivantes :

2	7	6
9	5	1
4	3	8

1	5	2
3	6	4
4	7	9

8	1	6
3	5	7
4	9	2

10	51	2
60	11	52
20	61	12

2	1
3	4

4	5	11	14
15	10	8	1
6	3	13	12
9	16	2	7

4	14	15	1
9	7	6	12
5	11	10	8
16	2	3	13

Exercice 13 (Table de 1 à n^2 *)**

Écrire un programme `creer_table(n)` qui construit une table de n lignes et n colonnes remplie des nombres de 1 à n^2, dans l'ordre.
Par exemple `creer_table(2)` retourne `[[1, 2], [3, 4]]`.
De même `creer_table(3)` retourne `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.

Exercice 14 (Tic Tac Toe **)**

Programmer le jeu du tic-tac-toe.
On pourra utiliser le code ci-dessous qui permet d'afficher une grille.


```
grille = [['o','*',' '], [' ','o', '*'], ['*', '*', 'o']]

def affichage(table):
    print("  A B C")
    print(" "+"-"*7)
    for i in range(3):
        ligne = str(i+1)
        for j in range (3):
            ligne = ligne + '|' + table[i][j]
        ligne = ligne + '|'
        print(ligne)
        print(" "+"-"*7)
```

Exercice 15 (Compléter un carré magique **)**

Écrire une fonction qui complète un carré magique incomplet de dimension 3×3. Ce programme retourne un message d'alerte si le carré ne peut être magique. La table de départ devra comporter au moins une ligne une colonne déjà remplie.
Utiliser votre programme pour compléter les carrés magiques suivant :

8	1	6
	5	

		4
	5	3
		8

		2
	5	7
		6

	5	
6	1	8

	51	
60	11	52

IX. Corrections des exercices

a) Exercices sur les tuples

Correction de l'exercice 1

```
# -*- coding: utf-8 -*-

from random import randint

def notes():
    """
    génère une série d'entre 3 et 10 notes sous la forme d'un tuple
    """
    serie = ()
    nbNotes = randint(3,10)
    for i in range(nbNotes):
        nvlleNote = randint(0,20)
        serie = serie + (nvlleNote,)
    return serie

def moyenne(serie):
    """
    calcul la moyenne d'une série de notes
    """
    long = len(serie)
    somme = 0
    for i in range(long):
        somme = somme + serie[i]
    return round(somme/long,2)

def note_min(serie):
    """
    trouve le plus petit élément d'une série de notes
    """
    candidat = serie[0]
    long = len(serie)
    for i in range(1,long):
        if candidat > serie[i]:
            candidat = serie[i]
    return candidat

def note_max(serie):
    """
    trouve le plus grand élément d'une série de notes
    """
    candidat = serie[0]
    long = len(serie)
    for i in range(1,long):
        if candidat < serie[i]:
            candidat = serie[i]
    return candidat

def statNotes(serie):
    """
    renvoie dans cet ordre
    la plus petite note
    la moyenne
    la plus grande note
    """
    noteMin = note_min(serie)
    moyenneNotes = moyenne(serie)
    noteMax = note_max(serie)
    return noteMin, moyenneNotes, noteMax
```

```python
# -*- coding: utf-8 -*-

from random import randint

def notes():
    serie = ()
    nbNotes = randint(3,10)
    for i in range(nbNotes):
        nvlleNote = randint(0,20)
        serie = serie + (nvlleNote,)
    return serie

notesNSI = notes()
notesFran = notes()
notesHG = notes()

def moyenne(serie):
    taille = len(serie)
    totalNotes = serie[0]
    #on commence pour i=1 car i=0 deja fait
    for i in range (1,taille):
        #on rajoute une note pour le total
        totalNotes = totalNotes + serie[i]
    #calcul de la moyenne
    moyenneNotes = totalNotes / taille
    #on renvoie les resultats
    return moyenneNotes

def bacOuPas(serieNSI,serieFran,serieHG):
    totalAuBac = 10*moyenne(serieNSI) + 5*moyenne(serieFran) + 3*moyenne(serieHG)
    moyenneAuBac = round(totalAuBac / (10+5+3),2)
    if moyenneAuBac >=10 :
        print("Bravo, avec une moyenne de",moyenneAuBac,"vous avez le BAC.")
    else :
        print("Ben non ! Avec une moyenne de",moyenneAuBac,"vous n'avez pas le BAC.")

#Bac ou pas
bacOuPas(notesNSI, notesFran, notesHG)
```

Correction de l'exercice 2 (Triplet Pythagoricien)

En rajoutant un petit compteur, on connait le nombre d'essais effectués par l'ordinateur.

```python
from random import randint

def tripletPytha():
    """
    renvoie un triplet Pythagoricien
    les trois valeurs sont données dans l'ordre croissant
    """
    nTests =0
    #cas impossible, on rentre dans la boucle tant que
    a, b, c = 1, 1, 1
    while a**2 + b**2 != c**2:
        #a est la plus petite longueur
        a = randint(1,99)
        #b est au moins aussi grande que a
        b = randint(a,99)
        #c la plus grande strictement que b
        c = randint(b+1,100)
```

```
        nTests = nTests+1
    print("Je suis ton ordi. Avec ton programme, j'ai eu besoin de",nTests,"essais pour y arriver.")
    #on sort de la boucle, on a notre triplet
    return a,b,c
```

Correction de l'exercice 3 (Triplet Pythagoricien **)

```
from random import randint

def TousLesTripletsPytha():
    """
    le programme teste toutes les combinaisons de a, b et c
    nombre entre 1 et 100 pour savoir quels combinaisons donnent
    un triplet pythagoriciens.
    Les nombres a, b et c sont données dans l'ordre croissant
    """
    nbTotal = 0
    triplets = ()
    for a in range(1,100):
        for b in range(a,100):
            for c in range(b+1,101):
                if a**2 + b**2 == c**2:
                    triplet = a,b,c
                    nbTotal = nbTotal + 1
                    #on rajoute le triplet au tuple triplets
                    triplets = triplets + (triplet,)
    print("Il y a donc",nbTotal,"de triplets Pythagoriciens composées de nombres entiers inférieurs a 100.")
    #on renvoie le tuple triplets contenant tous les tuples triplet
    return triplets
```

On obtient ainsi les 52 triplets pythagoriciens rangés dans l'ordre croissant avec des nombres entiers inférieurs ou égaux à 100.

Correction de l'exercice 4 (Triplet Pythagoricien ***)

```
from random import randint

def tripletPytha():
    nTests =0
    a, b, c = 1, 1, 1
    while a**2 + b**2 != c**2:
        a = randint(1,99)
        b = randint(a,99)
        c = randint(b+1,100)
        nTests = nTests+1
    return a,b,c,nTests

def tempsMoyen(nbRecherches):
    nTestsTotal = 0
    for i in range(nbRecherches):
        nTests = tripletPytha()[3]
        nTestsTotal = nTestsTotal + nTests
    nTestsMoyen = round( nTestsTotal / nbRecherches , 2)
    return nTestsMoyen
```

b) Exercices sur les tableaux

Correction de l'exercice 5 (Multiples, Diviseurs *)

```
def multiples1(n):
    """
    retourne un tableau des 10 premiers multiples non nuls de n
    """
    #on construit le tableau conteneur
    T = [0]*10
    #on le remplit
    for i in range(1,11):
        T[i-1] = i*n
    return T

def multiples2_v1(n):
    """
    retourne un tableau contenant tous les multiples non nuls de n
    inférieurs à 1000
    """
    #on calcule la taille du tableau
    taille = 1000//n
    #on construit le tableau conteneur
    T = [0]*taille
    #on le remplit
    for i in range(1,taille+1):
        T[i-1] = i*n
    return T

def multiples2_v2(n):
    """
    retourne un tableau contenant tous les multiples non nuls de n
    inférieurs à 1000
    dans cette version, on utilise la méthode append() donc on
    triche un peu car notre tableau n'est pas de taille fixe
    c'est donc une liste
    """
    T = []
    i = 1
    while i*n < 1000:
        T.append(i*n)
        i = i +1
    return T

def diviseurs(n):
    """
    retourne un tableau contenant tous les diviseurs de n
    cette fois-ci, on ne peut éviter la méthode append()
    car on ne connaît pas apriori le nombre de diviseurs d'un nombre
    """
    T = []
    diviseur = 1
    while diviseur <= n :
        if n%diviseur == 0 :
            #le reste de la division est 0
            T.append(diviseur)
        diviseur += 1
    return T
```

Correction de l'exercice 6 (Multiples, Diviseurs par compréhension **)

```
def multiples1(n):
    T = [i*n for i in range(1,11)]
    return T

def multiples2_v1(n):
    T = [i*n for i in range(1,1001) if i*n<1000]
    return T

def multiples2_v2(n):
    taille = 1000//n
    T = [i*n for i in range(1,taille+1)]
    return T

def diviseurs(n):
    return [d for d in range(1,n+1) if n%d==0]
```

Correction de l'exercice 7 (Multiplier un tableau *)

```
def produit(nombres,n):
    """
    renvoie un tableau contenant toutes les valeurs du tableau nombres
    mutlipliées par n
    """
    taille = len(nombres)
    T = [0]*taille
    for i in range(taille):
        T[i] = nombres[i]*n
    return T

#sinon le même mais par compréhension
def produit_v2(nombres,n):
    T = [element*n for element in nombres]
    return T
```

Correction de l'exercice 8 (Manipulations de tableaux ***)

```
from random import randint

def tableau_aleatoire():
    """
    génère un tableau de 20 valeurs pris aléatoirement entre 1 et 10
    """
    T = [randint(1,10) for i in range(20)]
    return T

def occurence(tab, n):
    """
    entrées :
        - tab est un tableau d'entiers pris entre 1 et 10
        - n est un entier pris entre 1 et 10
    rôle :
        la fonction renvoie le nombre d'occurence de n dans tab
    sortie :
        un entier
    """
    nb = 0
    for element in tab:
```

```
22          if element == n:
23              nb = nb + 1
24      return nb
25
26  def tableau_croissant(tab):
27      """
28      entrée :
29          tab est un tableau d'entiers pris entre 1 et 10
30      rôle :
31          la fonction renvoie un tableau contenant une et une seule fois
32          chaque éléments de tab, dans l'ordre croissant
33      sortie :
34          un tableau
35      """
36      T=[]
37      for i in range(1,11):
38          #on teste si i est présent dans tab
39          index = 0
40          present = False
41          while (index<len(tab)) and not present :
42              present = (tab[index] == i)
43              index = index + 1
44          if present :
45              #la valeur i est présente on la rajoute
46              T.append(i)
47      return T
48
49  def jointure(tab1,tab2):
50      """
51      entrées :
52          tab1 et tab2 deux tableaux d'entiers pris entre 1 et 10
53      rôle :
54          la fonction renvoie un unique tableau contenant une et une seule fois
55          chaque éléments de tab, dans l'ordre croissant
56      sortie :
57          un tableau
58      """
59      tab = tab1 + tab2
60      return tableau_croissant(tab)
```

Correction de l'exercice 9 (Manipulations de tableaux 2 *)**
Non corrigé.

c) Exercices sur les tables

Correction de l'exercice 10 (A faire à la main *)

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

1. Que retourne la commande `table[0][1]` ? `2`
2. Quelle commande retourne `8` ? `table[1][3]`
3. Quelle commande retourne `1` ? `table[0][0]`
4. Quelle commande retourne `4` ? `table[0][3]`

Correction de l'exercice 11 (Dimension d'un table *)

```
def dimensions(table):
    '''
    fonction qui retourne sous la forme d'un tuple les dimensions d'une table
    entrée :
        une table soit un tableau de tableaux
    sortie :
        un tuple
    '''
    nb_lignes = len(table)
    nb_colonnes = len(table[0])
    return (nb_lignes, nb_colonnes)
```

Correction de l'exercice 12 (Carré magique ***)

```
def somme_ligne(table,i):
    somme = 0
    for j in range(len(table)):
        somme = somme + table[i][j]
    return somme

def somme_colonne(table,j):
    somme = 0
    for i in range(len(table)):
        somme = somme + table[i][j]
    return somme

def somme_diagonale_hgbd(table):
    somme = 0
    for d in range(len(table)):
        somme = somme + table[d][d]
    return somme

def somme_diagonale_bghd(table):
    somme = 0
    for d in range(len(table)):
        somme = somme + table[d][-d]
    return somme

def carre_magique(table):
    nb_magique = somme_ligne(table,0)
    reponse = True
    #on teste toutes les lignes
    i = 0
    while i<len(table) and reponse == True:
        reponse = (nb_magique == somme_ligne(table,i))
        i = i + 1
    #on teste toutes les colonnes
    i = 0
    while i<len(table) and reponse == True:
        reponse = (nb_magique == somme_colonne(table,i))
        i = i + 1
    #on teste les deux diagonales
    if reponse == True:
        reponse == (nb_magique == somme_diagonale_hgbd(table))
    if reponse == True:
        reponse == (nb_magique == somme_diagonale_hgbd(table))
    return reponse
```

Correction de l'exercice 13 (Table de 1 à n^2 ***)

```python
def creer_table(n):
    """
    génère une table de dimension nxn
    contenant dans l'ordre, tous les nombres de 1 à n**2
    """
    T = []
    for i in range(0,n):
        #création de i-ème ligne
        ligne = [0]*n
        #remplissage de la ligne
        for j in range(0,n):
            ligne[j] = i*n + j + 1
        #on rajoute la ligne à la table
        T.append(ligne)
    return T
```

Correction de l'exercice 14 (Tic Tac Toe **)**
Non corrigé.

Correction de l'exercice 15 (Compléter un carré magique **)**
Non corrigé.