**RND. Types. Construits**

**I/ Introduction**

Dans la partie précédente, différents types de valeurs ont été présentées comme *int*, *bool*, *float* : ce sont des types **simples.** Il existe également le type *string*, qui est composé d’une **chaîne de caractères** et permet de traiter du texte et également assimilé à un type simple.

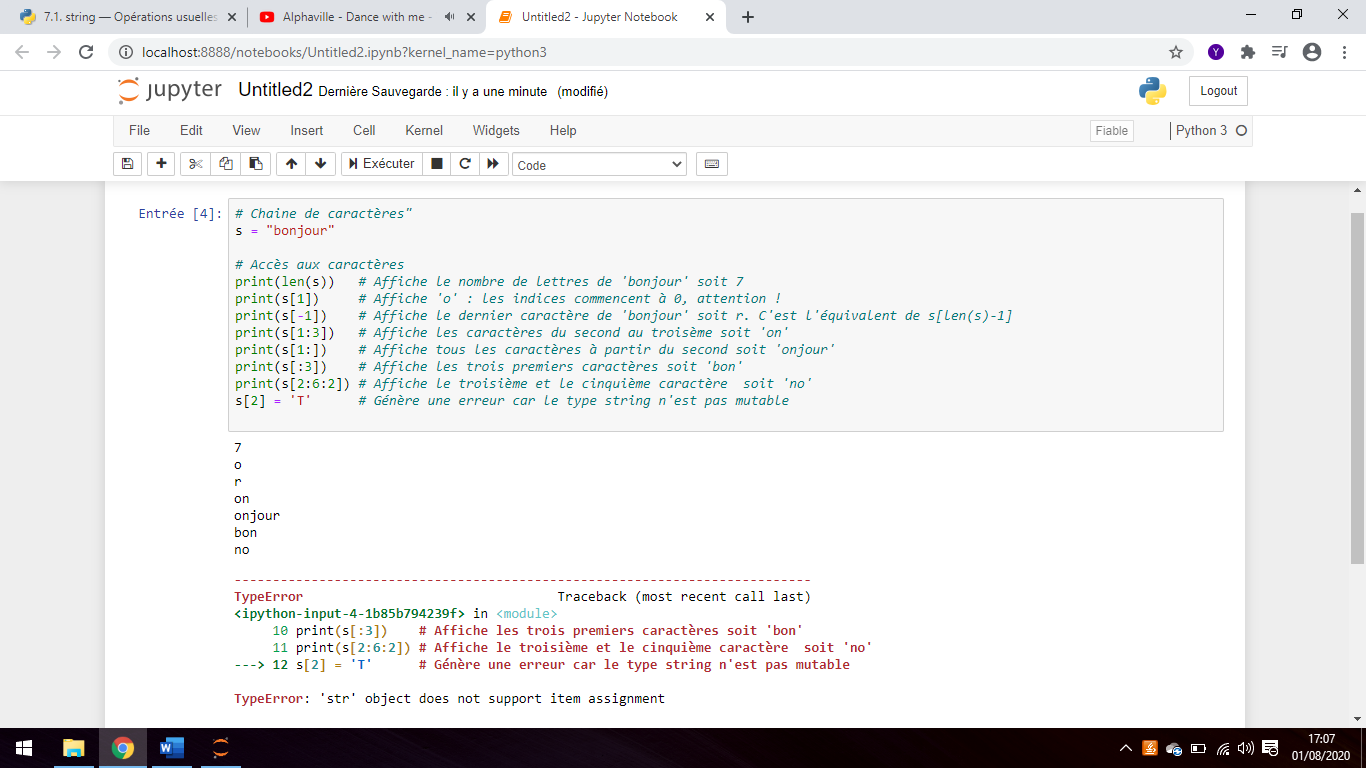
Rappel : chaque caractère correspond à un nombre attribué en fonction de la norme choisie (Unicode, ASCII etc.). **Un mot n’a aucun sens pour un ordinateur**, il est traité comme une suite de caractères indépendants.

On peut ainsi associer une **valeur** à une **variable**. Cela est suffisant si l’on n’utilise que peu de données mais s’il devient important, il faudra les « regrouper » par exemple sous forme de tableaux. Ainsi, **une seule variable** peut définir un **ensemble de valeurs**.

**II/ Chaînes de caractères**

Le langage Python propose le type *string* pour gérer les chaînes de caractères. Un grand nombre de ses **méthodes** sont communes aux types construits. On notera qu’une chaîne de caractères n’est **pas** **mutable**, ce qui signifie que l’on ne peut **pas** la **modifier**.

Voici quelques méthodes d’accès aux caractères d’un type *string* :



Il existe bien entendu bien plus de méthodes pour le type *string*. Ne pas hésiter à aller consulter la documentation officielle du langage Python ou de consulter le livre de *Gérard Swinnen*.

Documentation : <https://docs.python.org/fr/3.8/library/string.html>

*A noter : une* ***méthode*** *(ou fonction-membre) est une fonction qui n’est valable que pour le type en question. Cela est une des bases de la programmation objet qui sera étudiée en Terminale.*

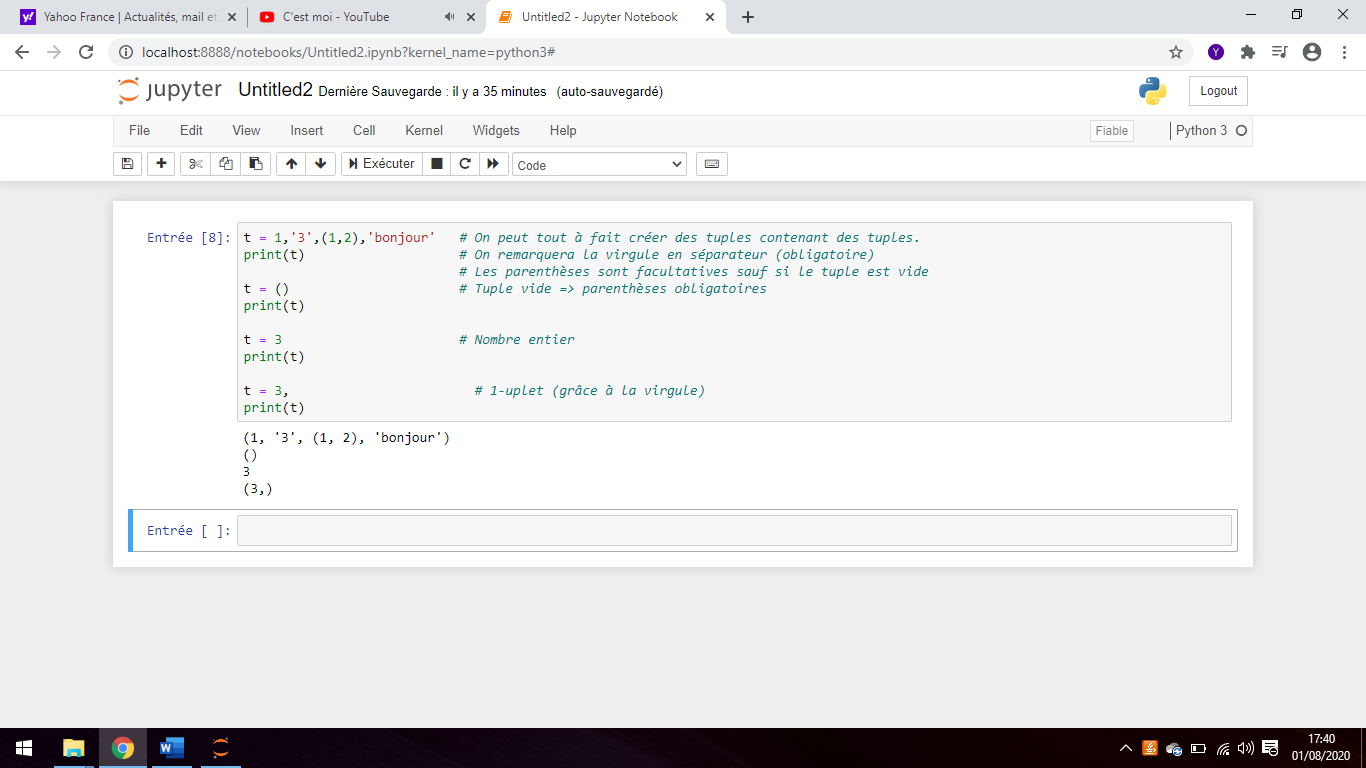
**III/ N-uplets**

**Un n-uplet** est une **suite ordonnée d’éléments** qui peuvent être **de n’importe quel type**. Tout comme le type *string*, il n’est **pas mutable**.

Exemple : t = 1,2,5,’yop’ crée un tuple de quatre éléments.

Le langage Python utilise les parenthèses pour indiquer qu’il s’agit d’un type tuple.

Initialisation d’un type *tuple*:



Les **méthodes d’accès aux éléments** sont **les mêmes** que celles du type *string*.

**IV/ Listes**

**1/ Définition et construction**

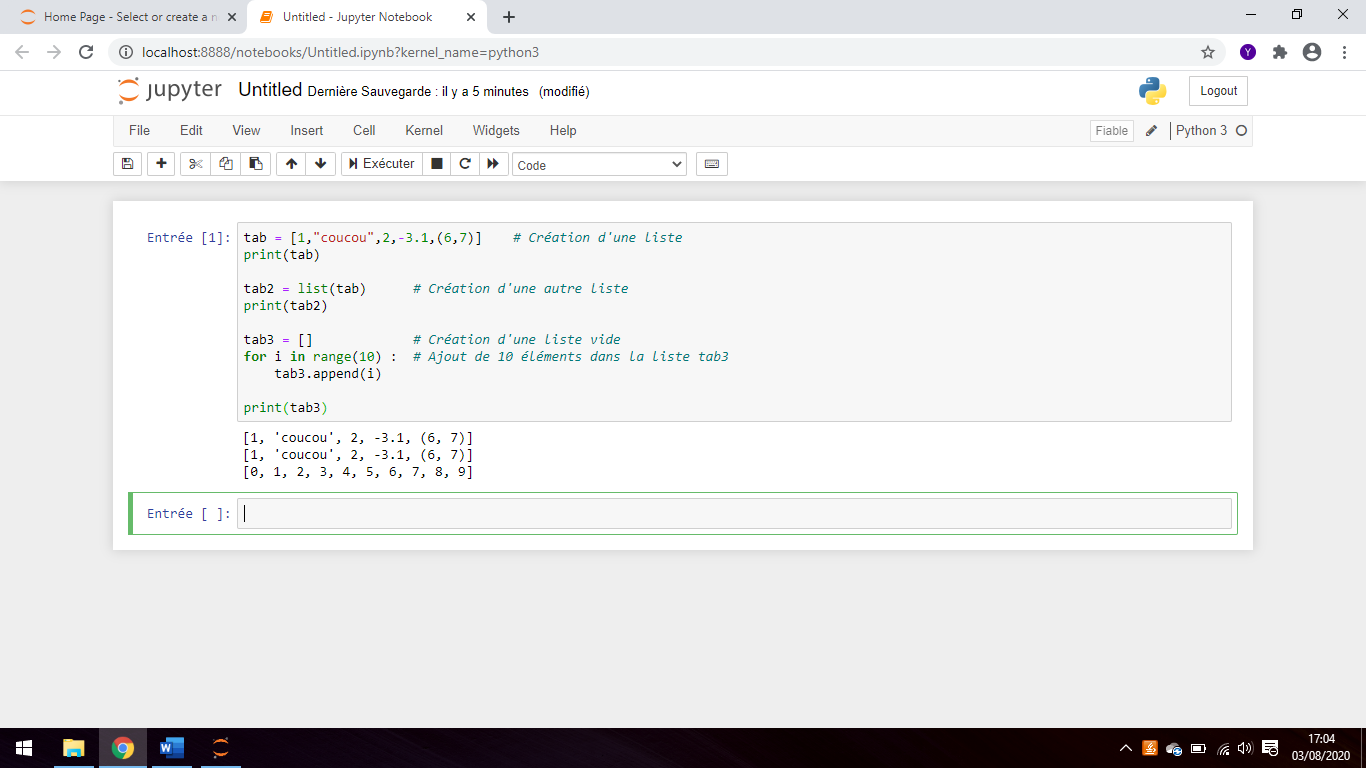
Une liste est une **suite d’éléments ordonnée** (comme le tuple) mais **mutable**. En langage Python, ce sont les crochets qui permettent de déterminer un objet de type *list*. Là encore, les types de chaque élément peuvent différer mais cela est rarement utile.

On peut tout à fait créer des « listes de listes » ce qui est très utile pour parcourir les pixels d’une image par exemple ou de déterminer un ensemble de points munis de coordonnées du type (y , f(y)) où *f* est une fonction mathématique.

Exemples :

* Liste1 = [3,6, « bonjour »,-5.4] est une liste.
* Liste2 = [[5,4],[2,-3],[-4,7]] est une liste de listes.

Initialisation d’un type *list* :



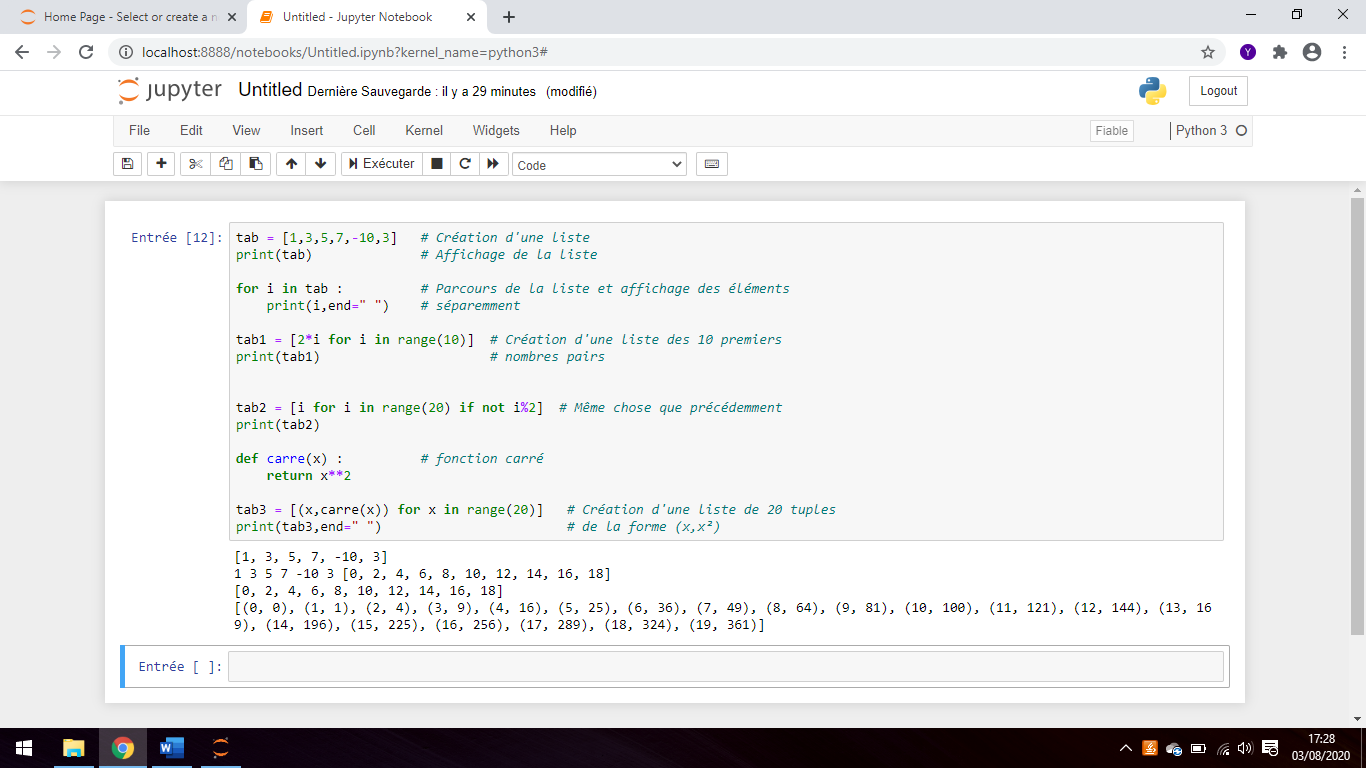
**2/ Construction par compréhension**

La construction d’une liste par **compréhension** est très utile par sa concision de code. L’instruction est de la forme

[ *expression(i)* for i in *objet* *condition*] .

**Expression(i)** dépend de i (cela peut être une fonction), **objet** peut être un objet itérable ou une instruction à partir de range(), **condition** est un test avec l’instruction *if* mais est facultative.

A noter : il peut y avoir des boucles *for* imbriquées.

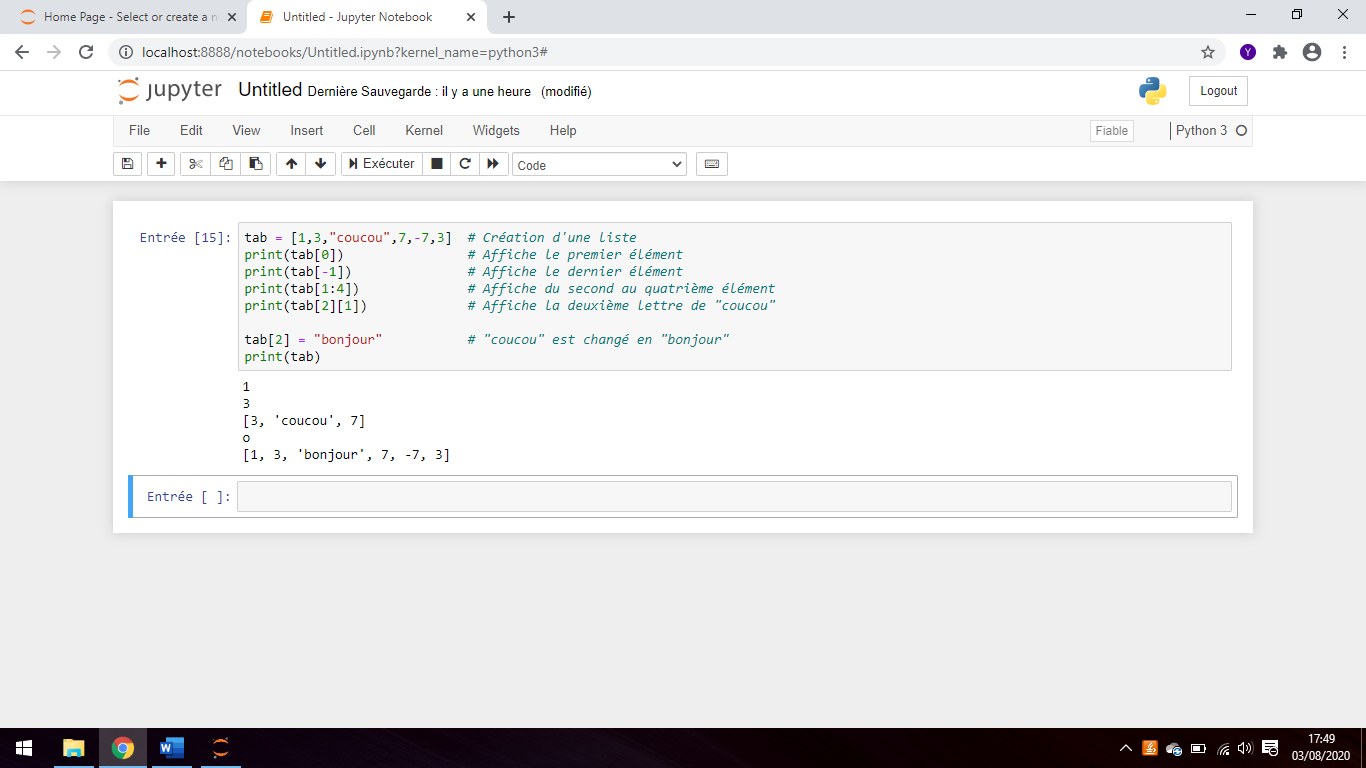
Compréhension de *list* :

Ces exemples permettent de comprendre que les listes sont très adaptées à la manipulation d’éléments de type numérique.

**3/ Accès aux éléments**

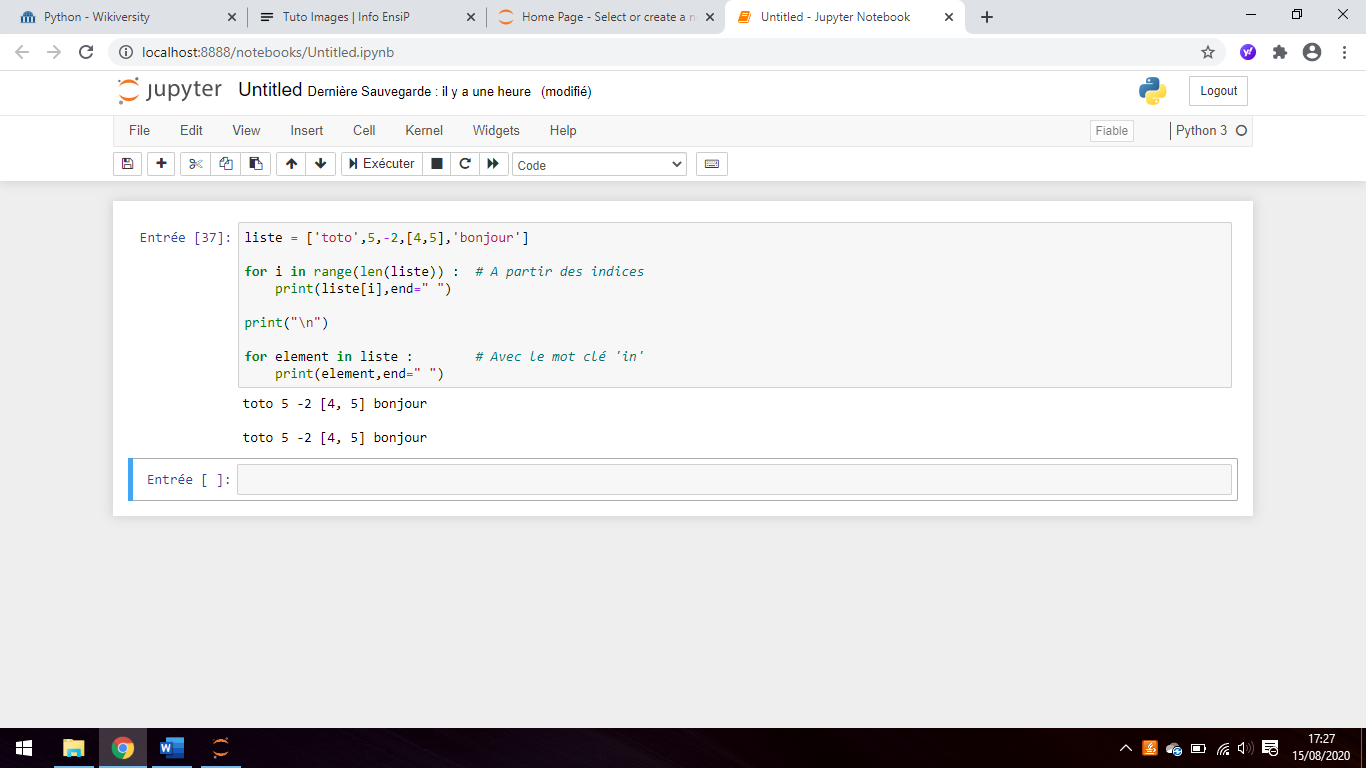
Comme pour les type *string* et *tuple*, on accède aux éléments par l’opérateur [*indice***]**. Avec une liste, on peut également les modifier directement à partir de leur indice.

Accès aux éléments et modification de liste :



On peut **parcourir une liste** de deux façons.

* A partir des indices
* En utilisant le mot clé **in** du langage Python

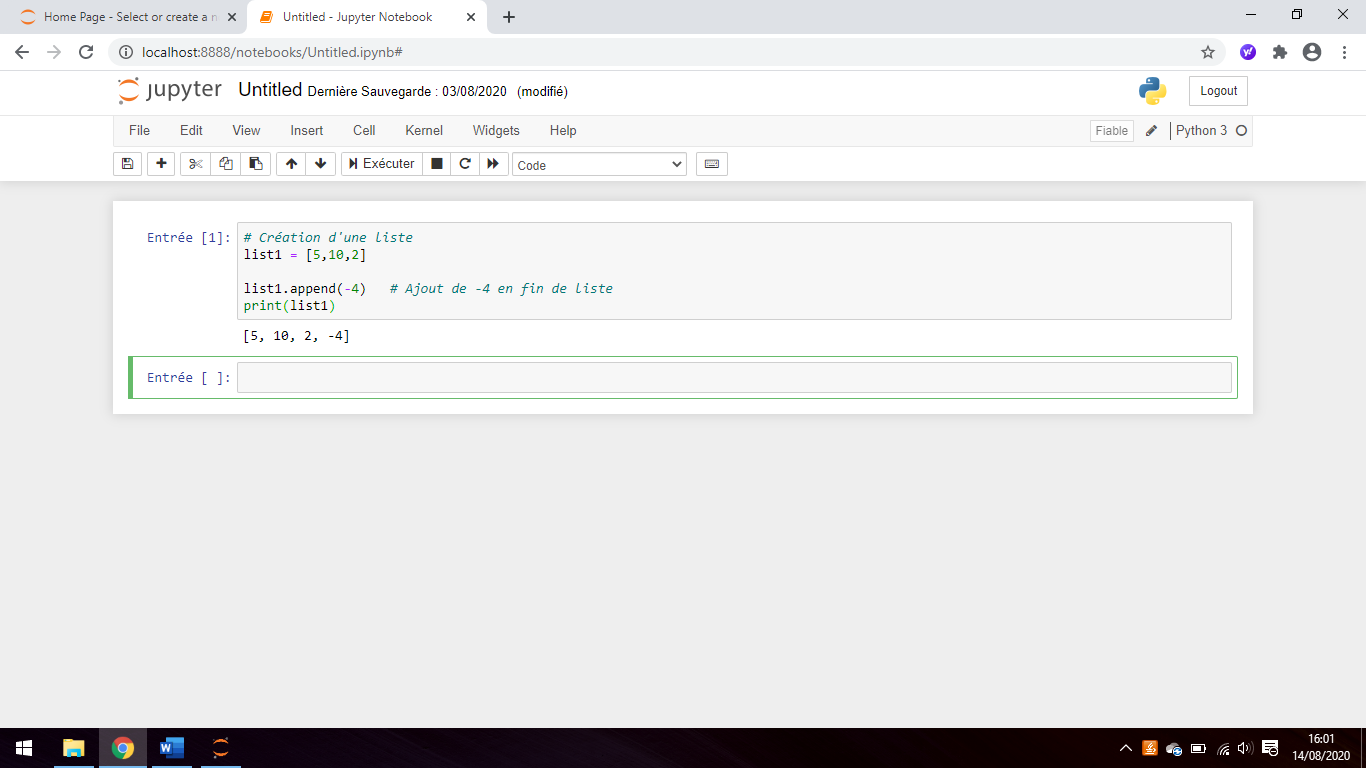


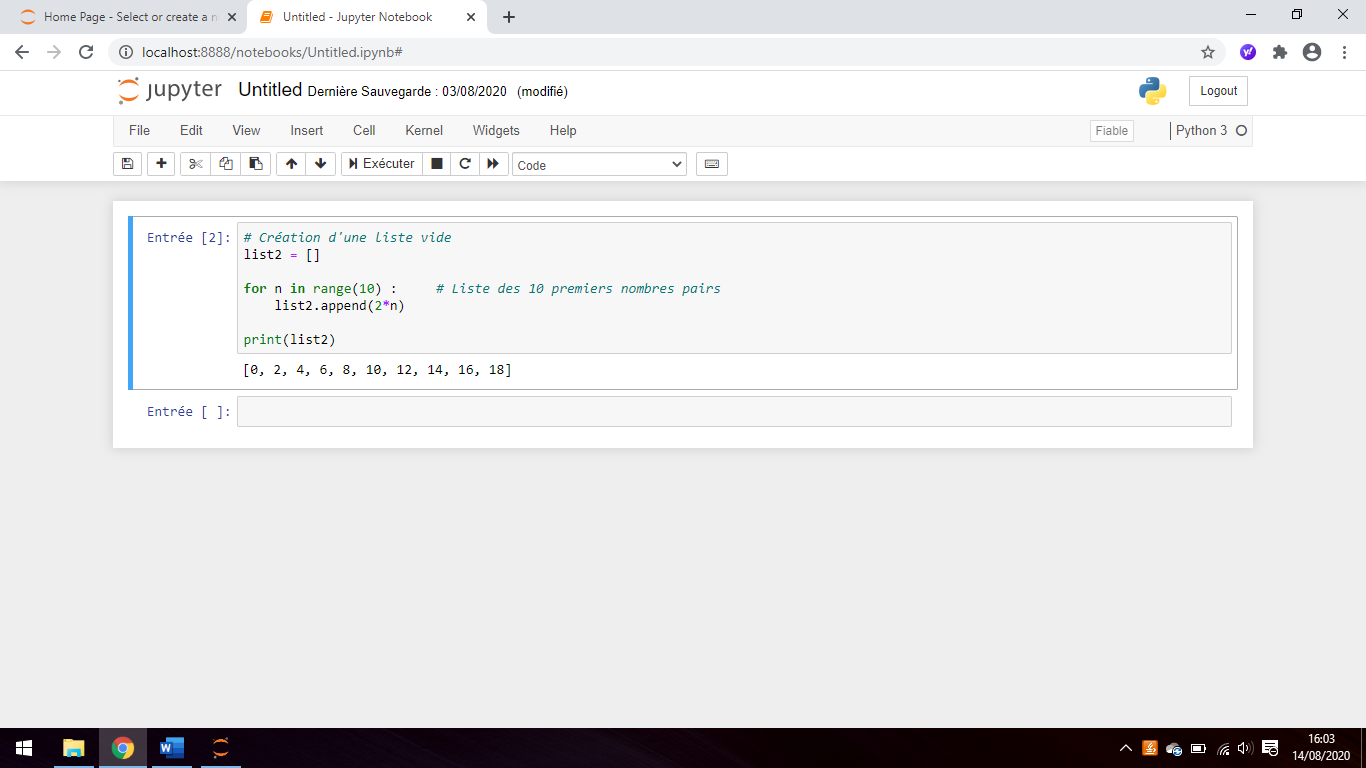
**4/ Méthodes des listes**

Voici quelques **méthodes** de l’objet *list* qui sont **à connaître** : append(*valeur*), insert(*index*,*valeur*), remove(*valeur*), pop(*index*), index(*valeur*), reverse(), sort().

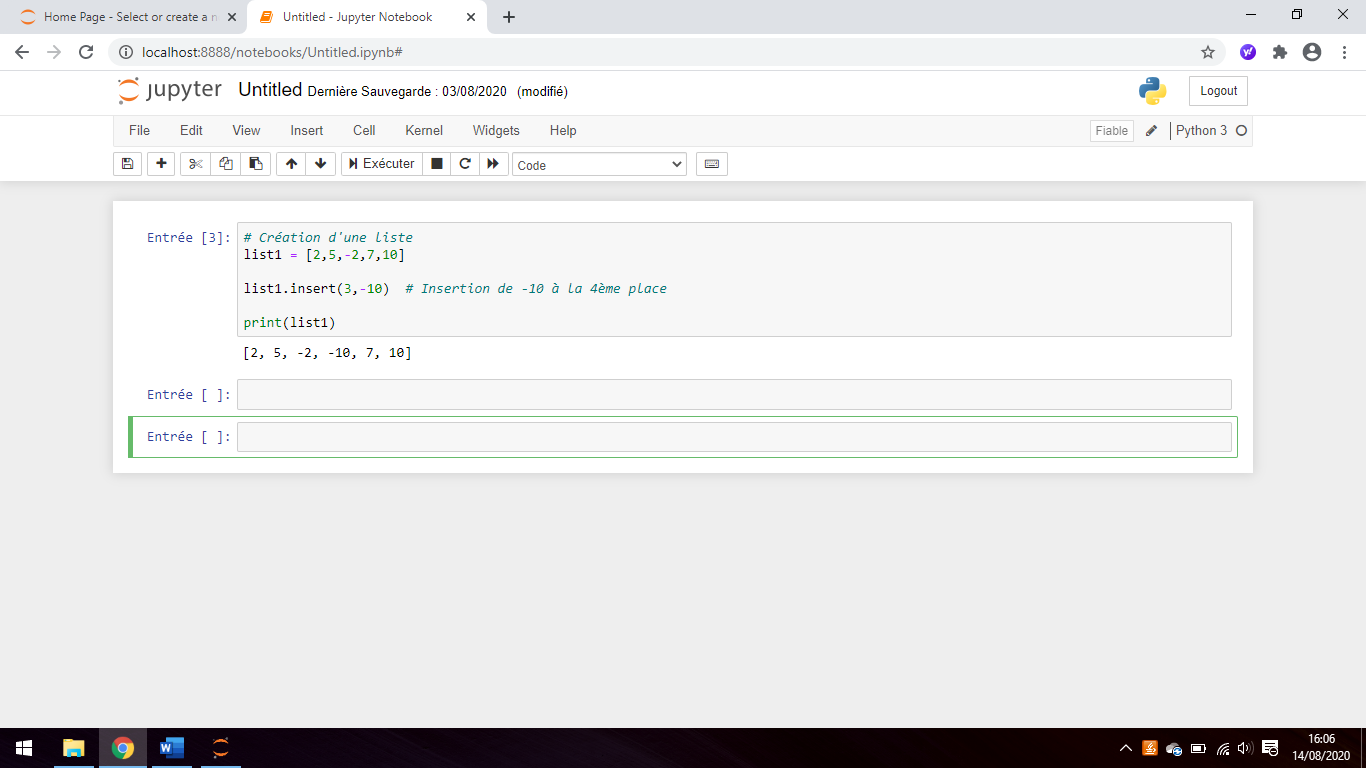
A noter : ces **méthodes** ne **renvoient** **aucune valeur** : ce qui signifie que la liste initiale (avant modification) est perdue.

La méthode **append(*valeur*)** de *list* :

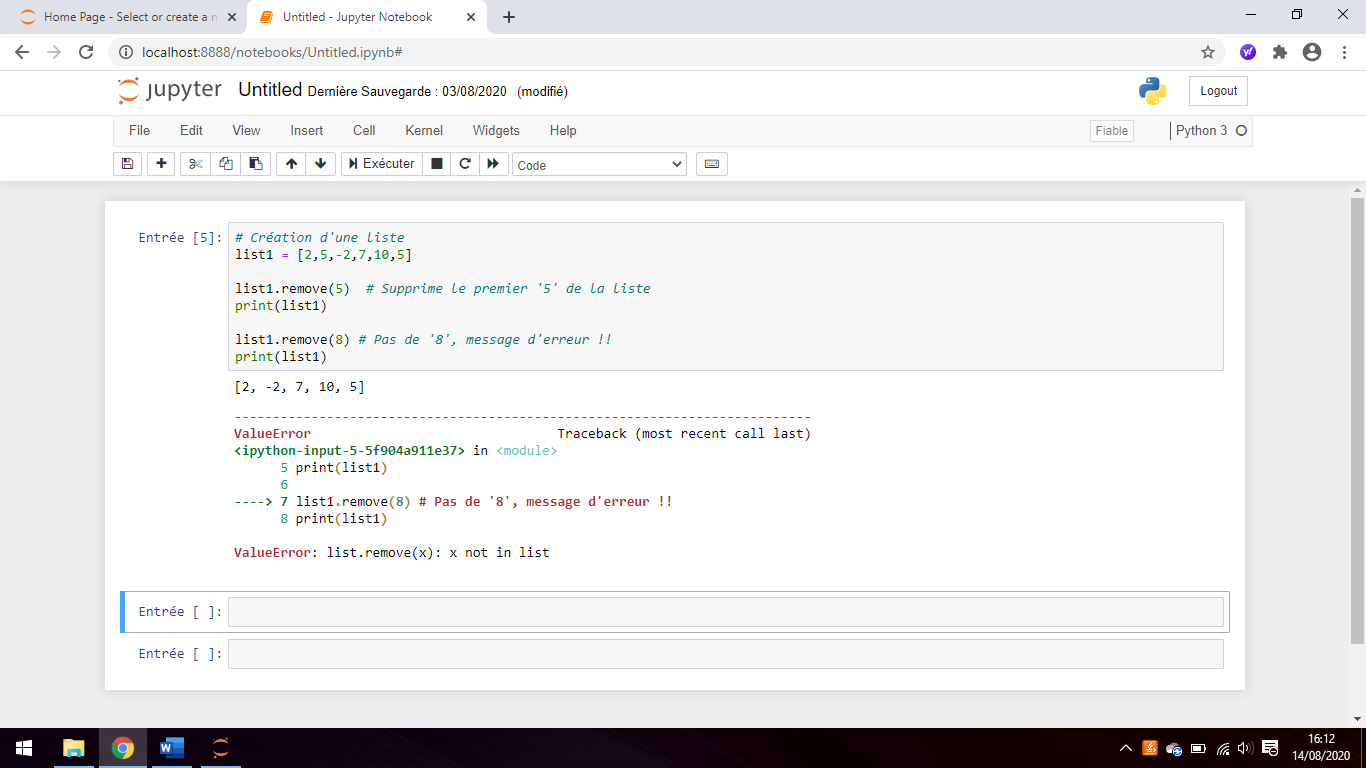




La méthode **insert(index,*valeur*)** de *list* :



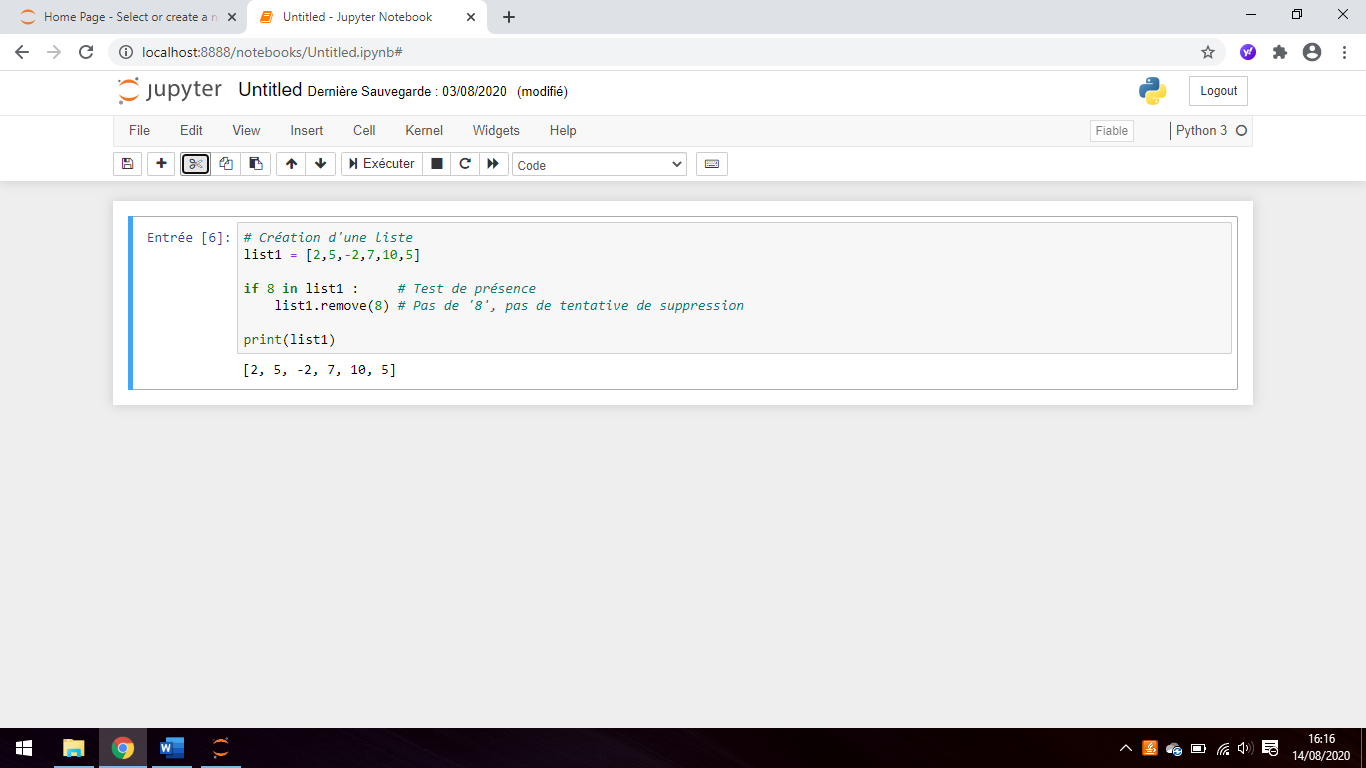
La méthode **remove(*valeur*)** de *list* :



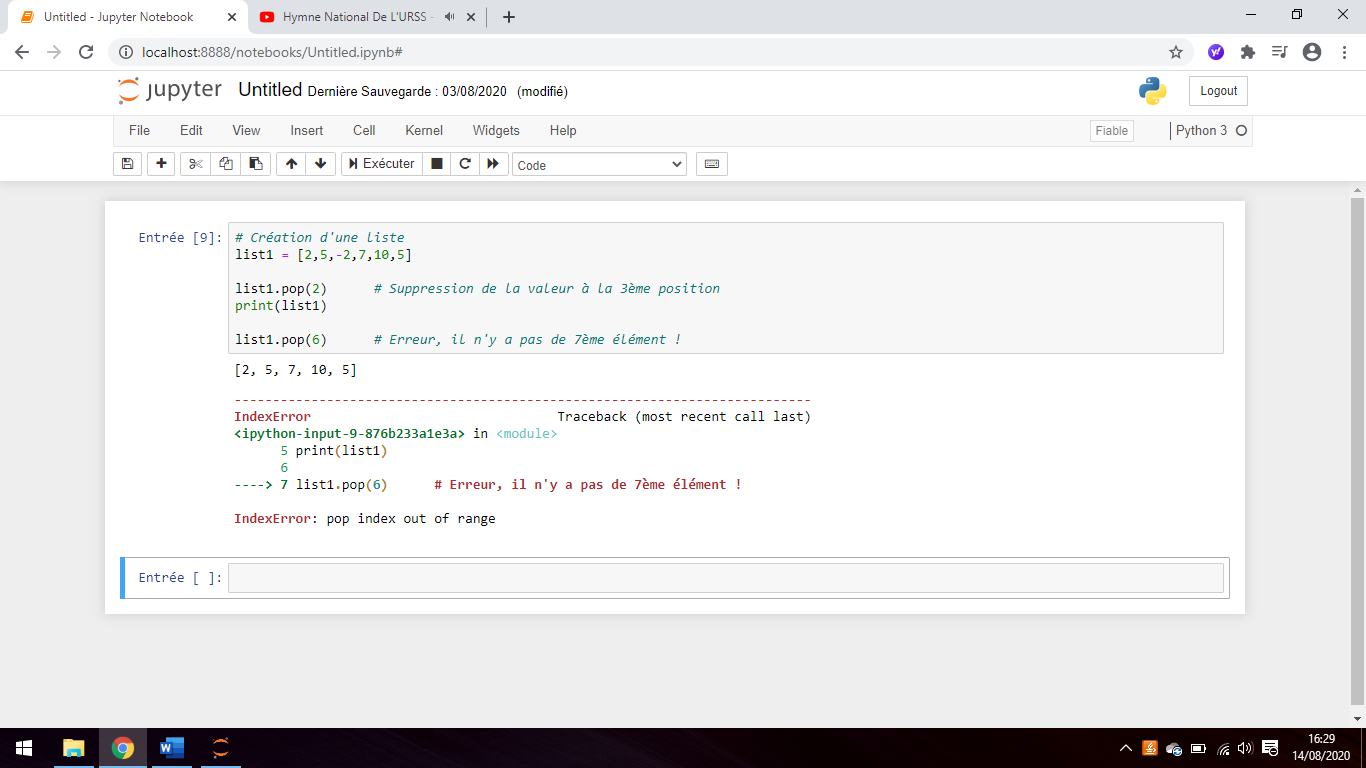
On sera attentif au fait que cette remove(valeur) ne supprime que la **première occurrence de la valeur (**s’il y a).

**Attention** : un message d’erreur apparaît si la valeur n’est pas dans la liste et stoppe le programme.

On peut utiliser un test pour éviter cela.

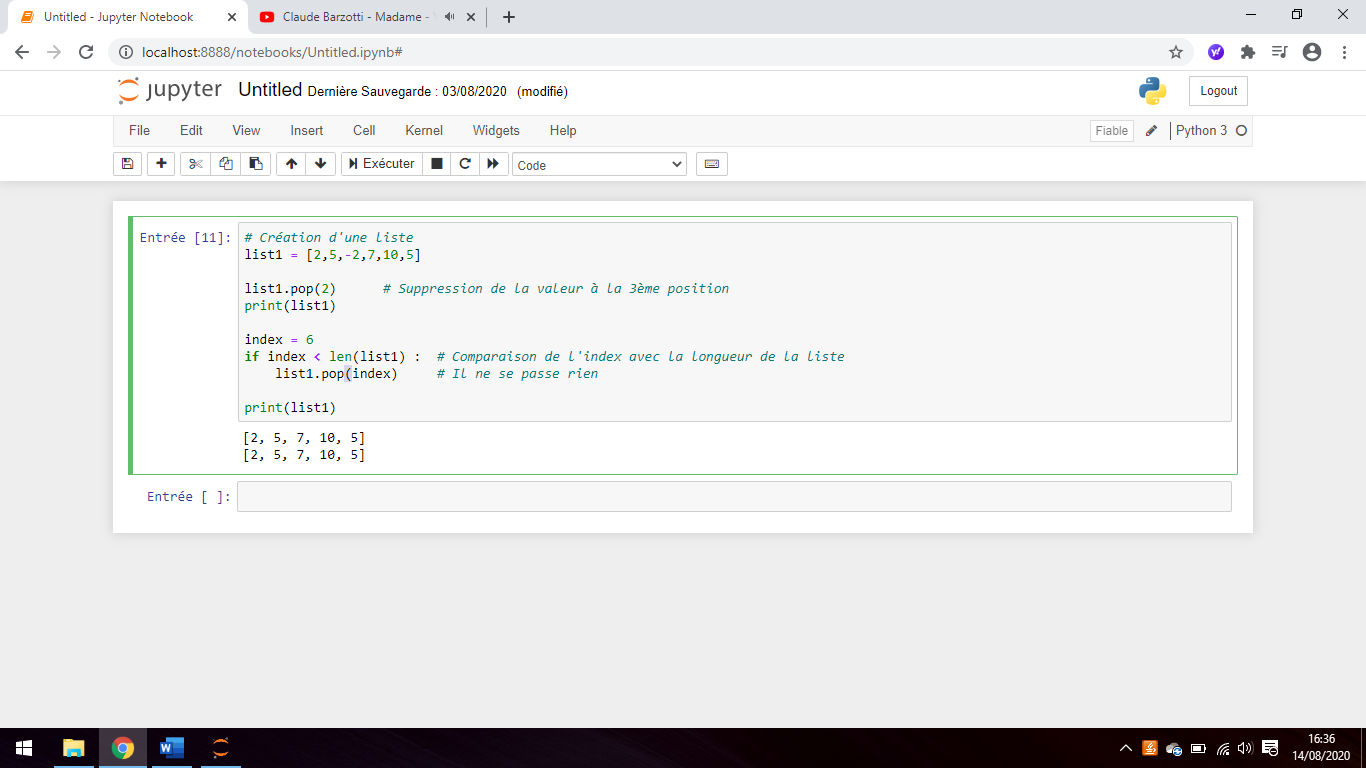


La méthode **pop(*index*)** :



**Attention** : un message d’erreur apparaît si la l’index dépasse la position du dernier élément de la liste et stoppe le programme

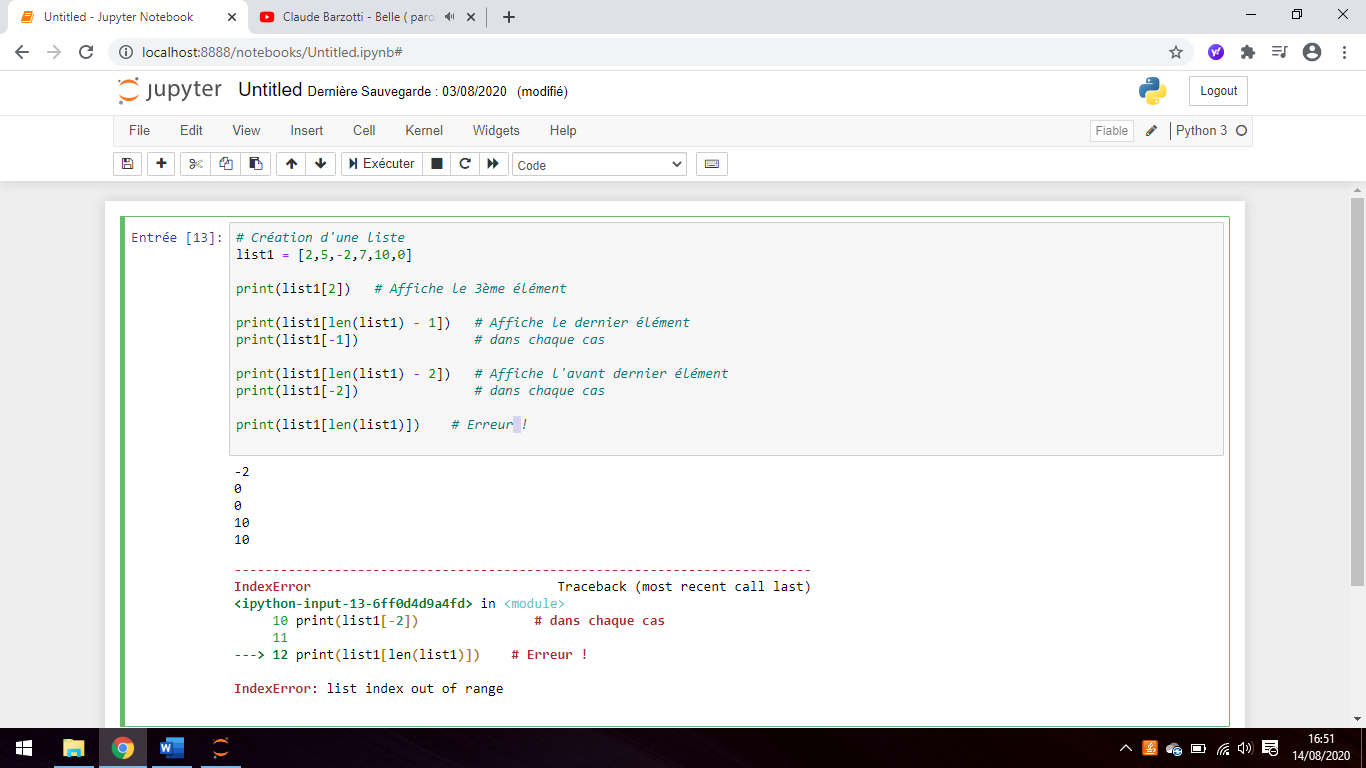
On peut utiliser un test pour éviter ce cas là :



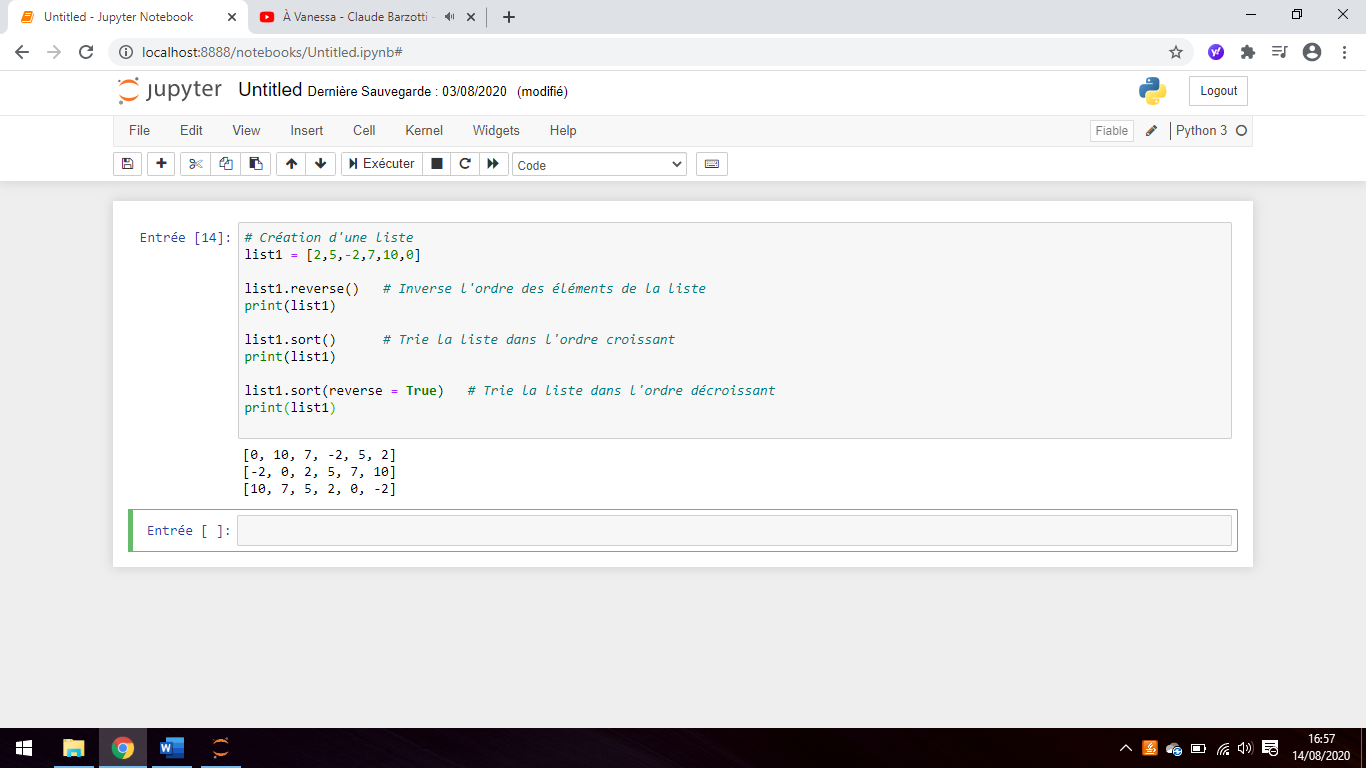
**A savoir** : la fonction **len(*tab*)** donne le nombre d’éléments de l’objet *tab*. Elle est très utile car la plupart du temps, on ne connaît pas la taille d’une *list* (chargement de données d’un fichier par exemple).

**Attention** : les indices commençant à 0, on accède au dernier élément d’une *list* par l’instruction tab[len(tab) – 1].

**A savoir** : particularités du langage Python dans l’accès aux éléments :



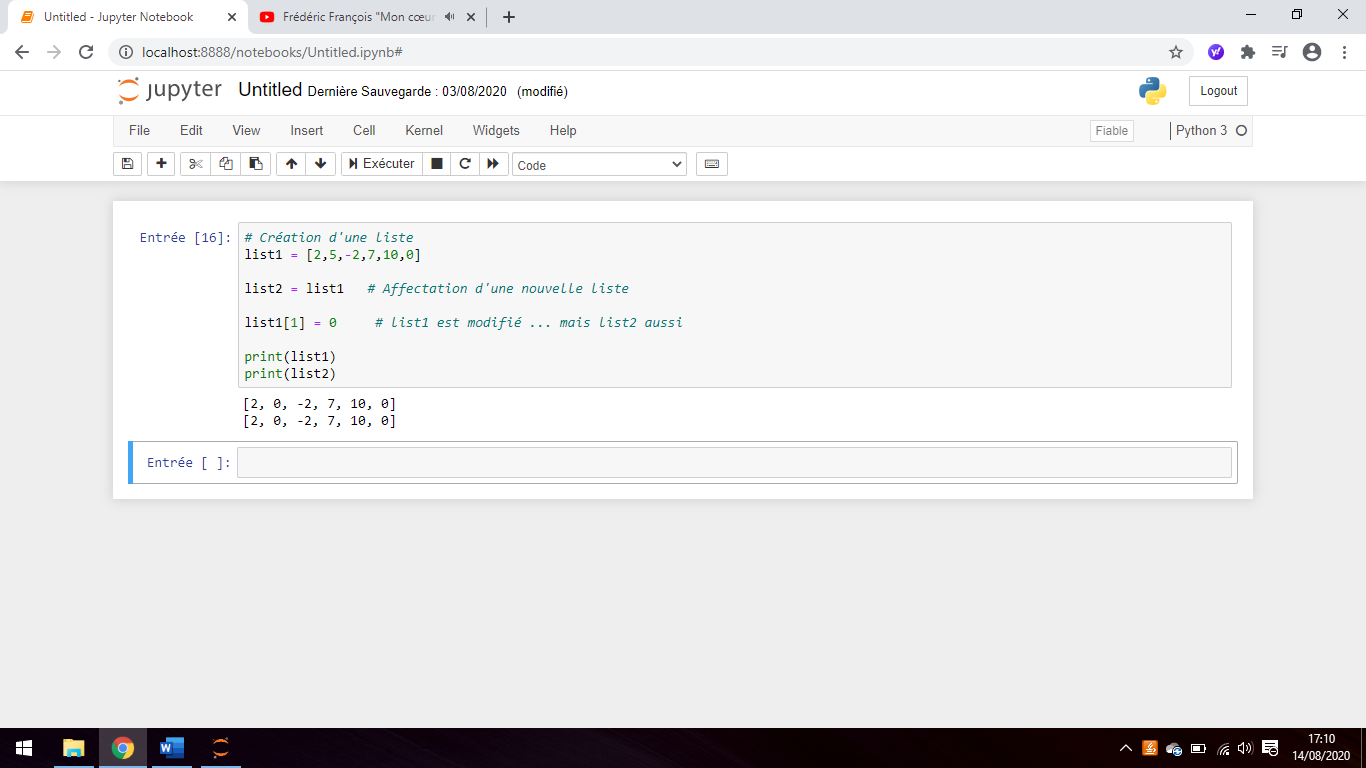
Les méthodes **reverse()** et **sort()** :



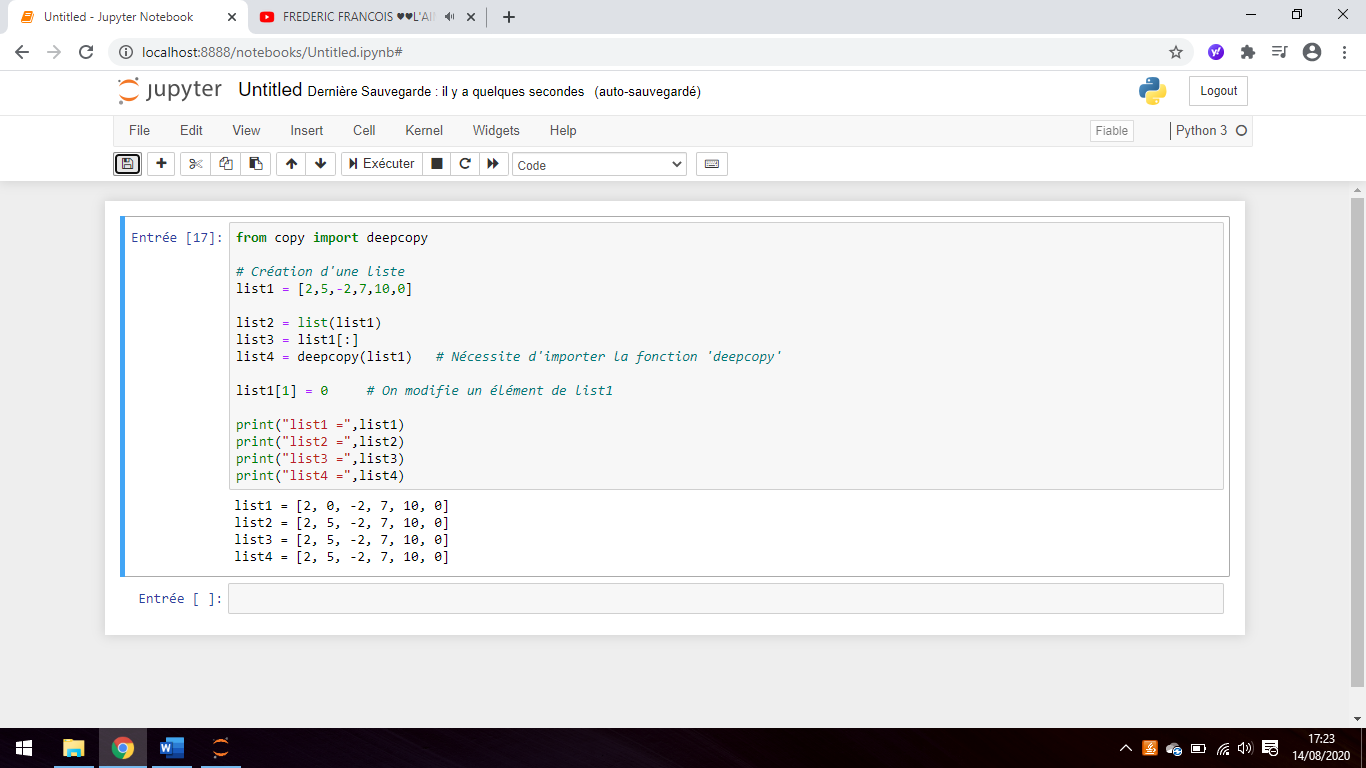
**5/ Copie de listes**

Il y a une différence très importante entre les objets de type *list* et ceux de type *string* ou *tuple* : il est possible de **modifier** **un élément** d’un type ***list*** par **affectation**.

**Attention** : **une affectation entre deux listes ne crée pas une nouvelle liste**. Ce qui a des conséquences si l’on **modifie un élément**, il le sera pour les **deux listes**.



Pour éviter ces désagréments, on peut utiliser l’une de ces méthodes (il y en a d’autres) :



Plus de détails sur les références partagées à ce lien : <https://www.youtube.com/watch?v=XkFYa8genoI>

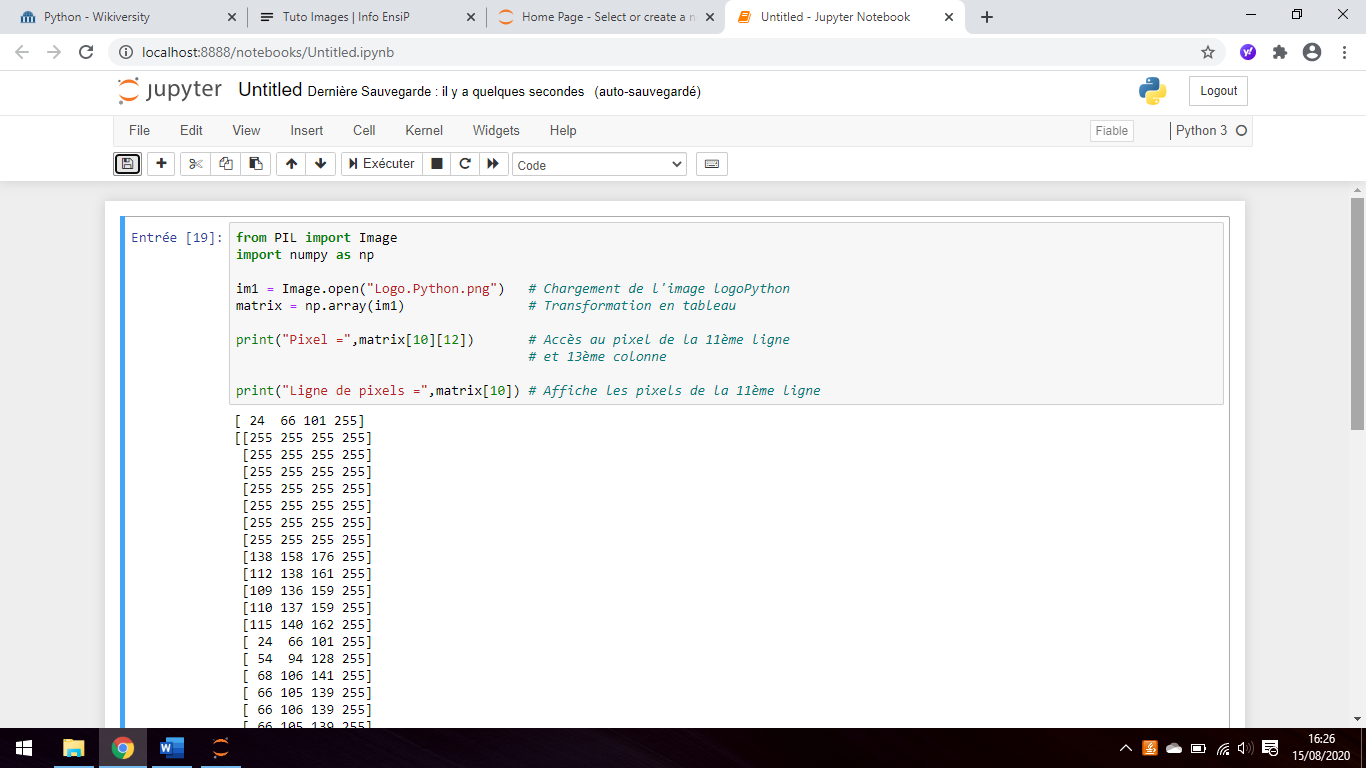
Auteur : Cours Python 3, durée : 10 min 56 sec

Remarque : en pratique, il est très rare d’avoir à affecter directement une liste à une autre sans en modifier ses valeurs.

**6/ Tableaux et matrices**

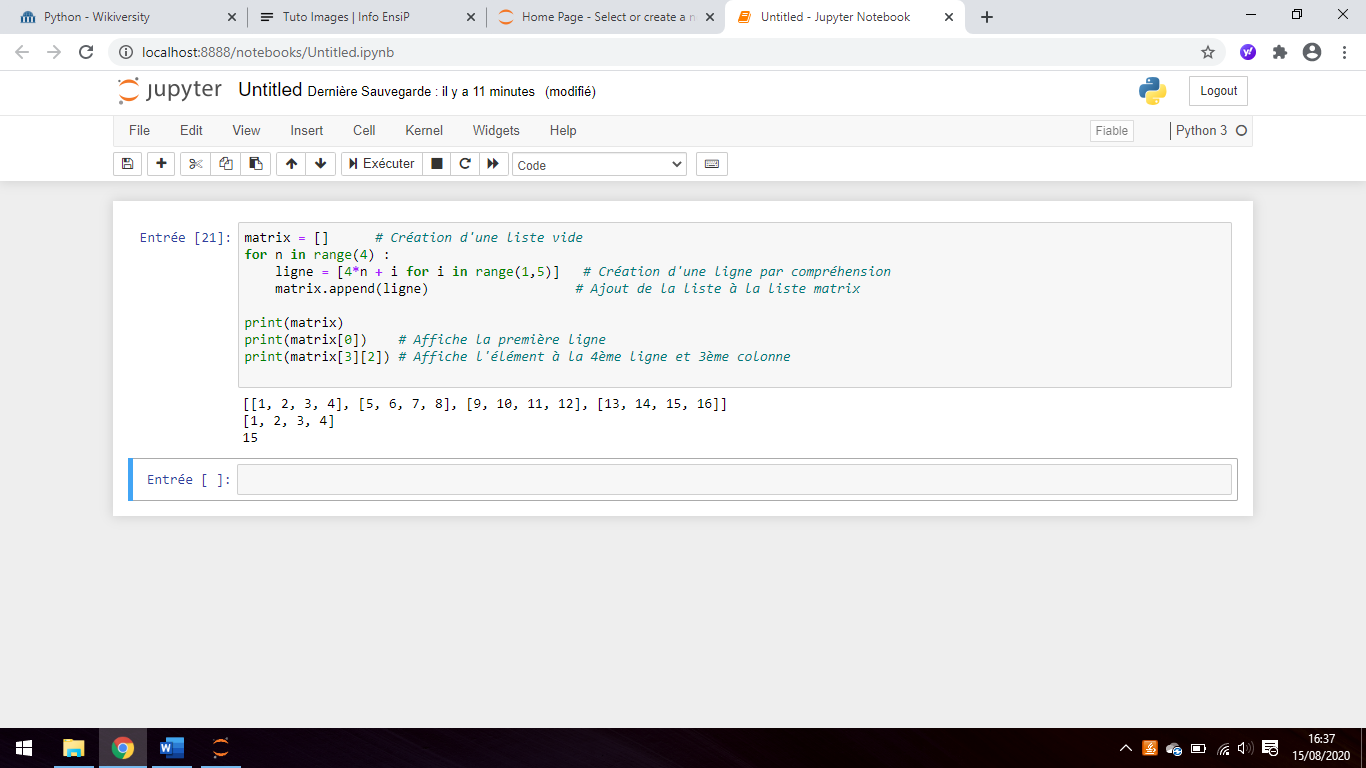
Une liste peut être composée d’éléments de différents types y compris de listes. Une liste composée de *n* listes de longueurs *p* est appelée **matrice** (*n*,*p*) avec *n* lignes et *p* colonnes.

Exemple : une matrice peut être utilisée pour représenter une image : chaque **élément** représente un pixel et chaque **liste** une ligne de pixels.

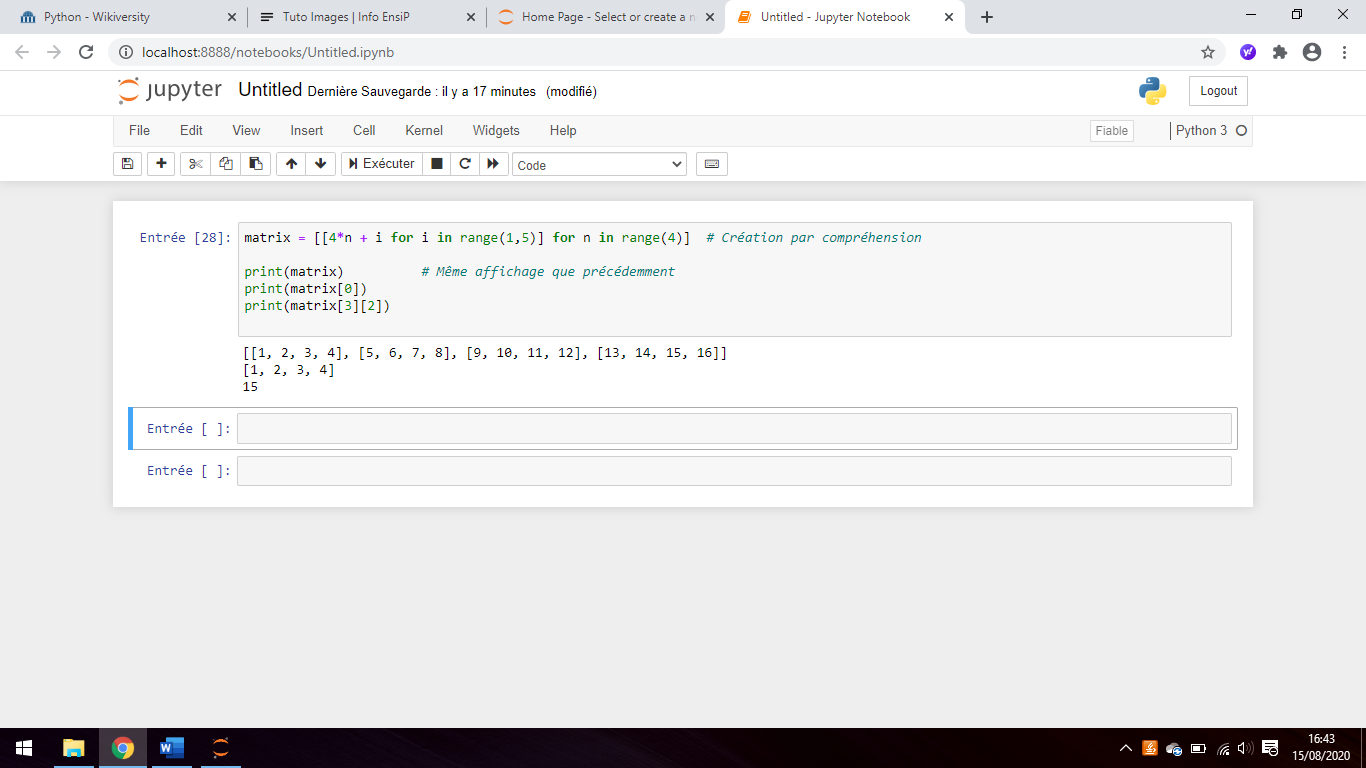


Remarque : il s’agit ici d’une image *png* qui prend en charge la transparence, chaque pixel est donc composé de 4 octets : rouge, vert, bleu et alpha avec des valeurs comprises entre 0 et 255.

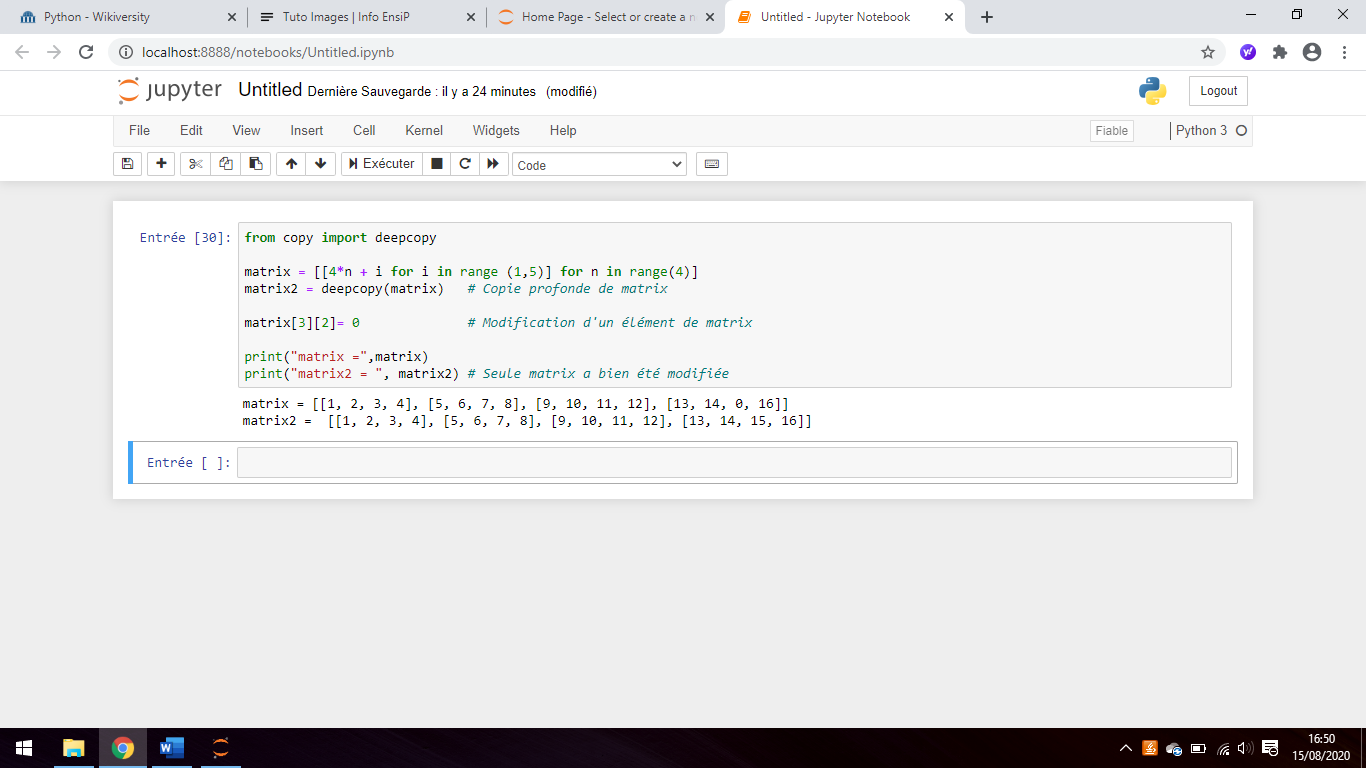
Exemples de création de matrices :



**Méthode de construction par compréhension de matrice à connaître :**



Pour **copier** une **matrice** sans **effet de bord**, on utilise la méthode *deepcopy* :



**V/ Dictionnaires**

**1/ Définition et construction**

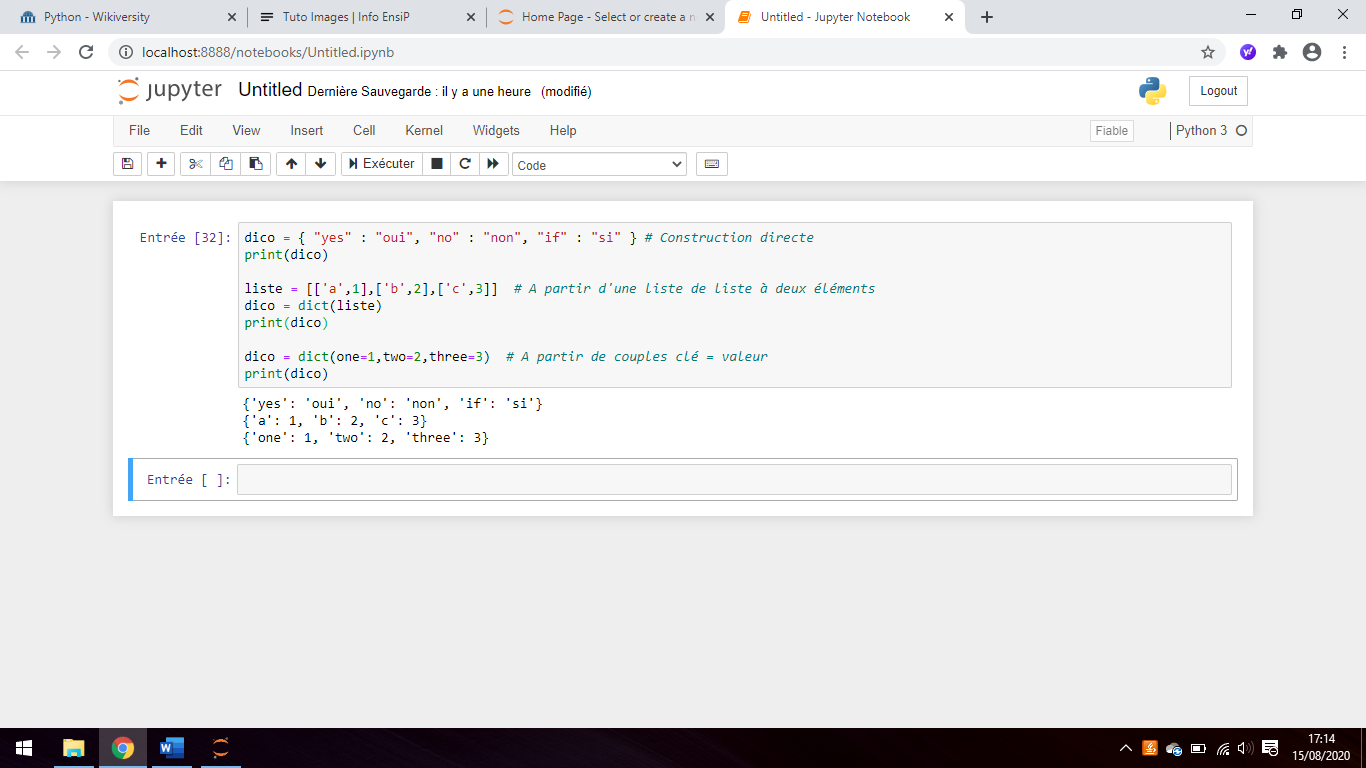
Un dictionnaire, un objet de type *dict*, ressemble à une liste de liste de deux éléments.

* Le premier élément est appelé **clé** et est de *type* numérique ou string la plupart du temps (objet **non mutable**).
* Le second élément est appelé **valeur** et est **relié à sa clé**. Il peut être de tout type.

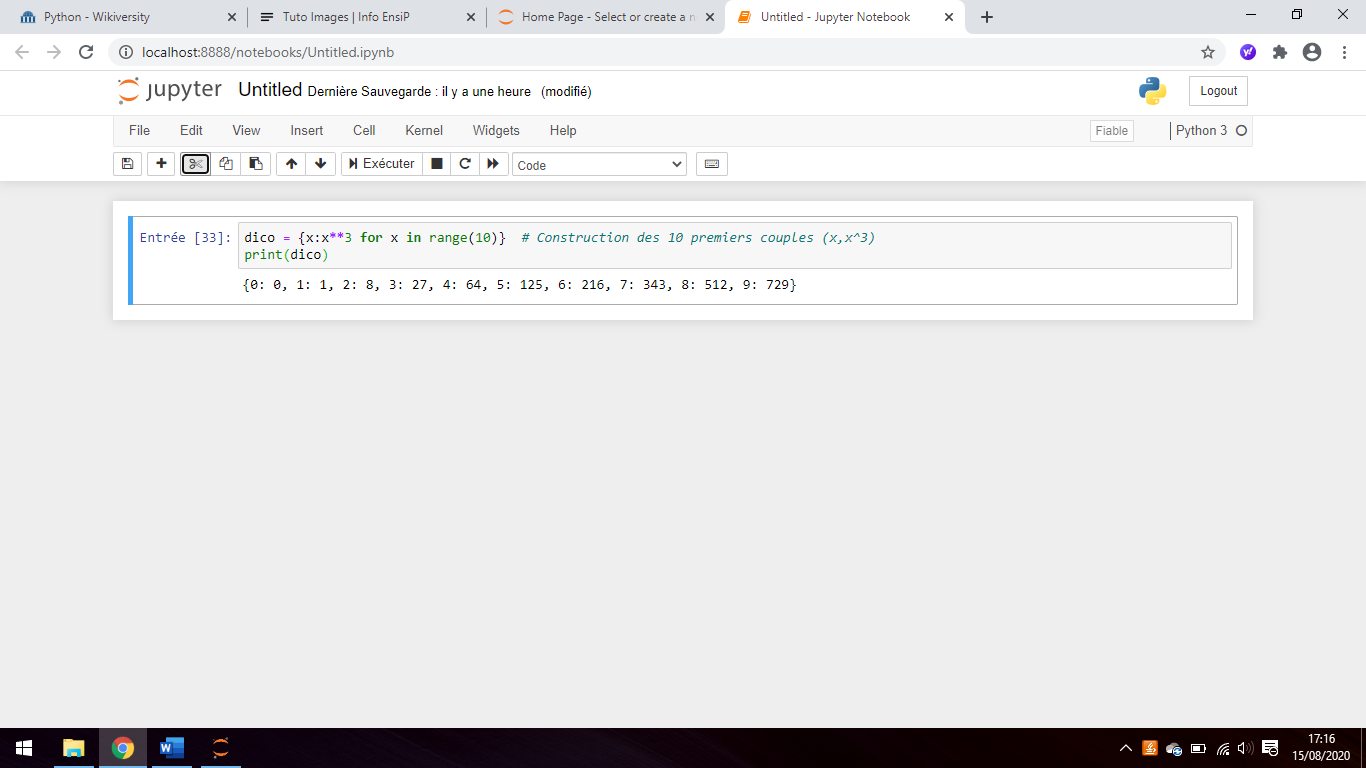
Sa construction est proche de cette d’une liste de liste à deux éléments et le langage Python permet la même souplesse.

On utilise les accolades {} pour délimiter le dictionnaire et les deux points : pour séparer la clé de la valeur.

Exemples de construction d’un dictionnaire :



**La méthode de construction par compréhension est bien sûr disponible (et est à connaître) :**

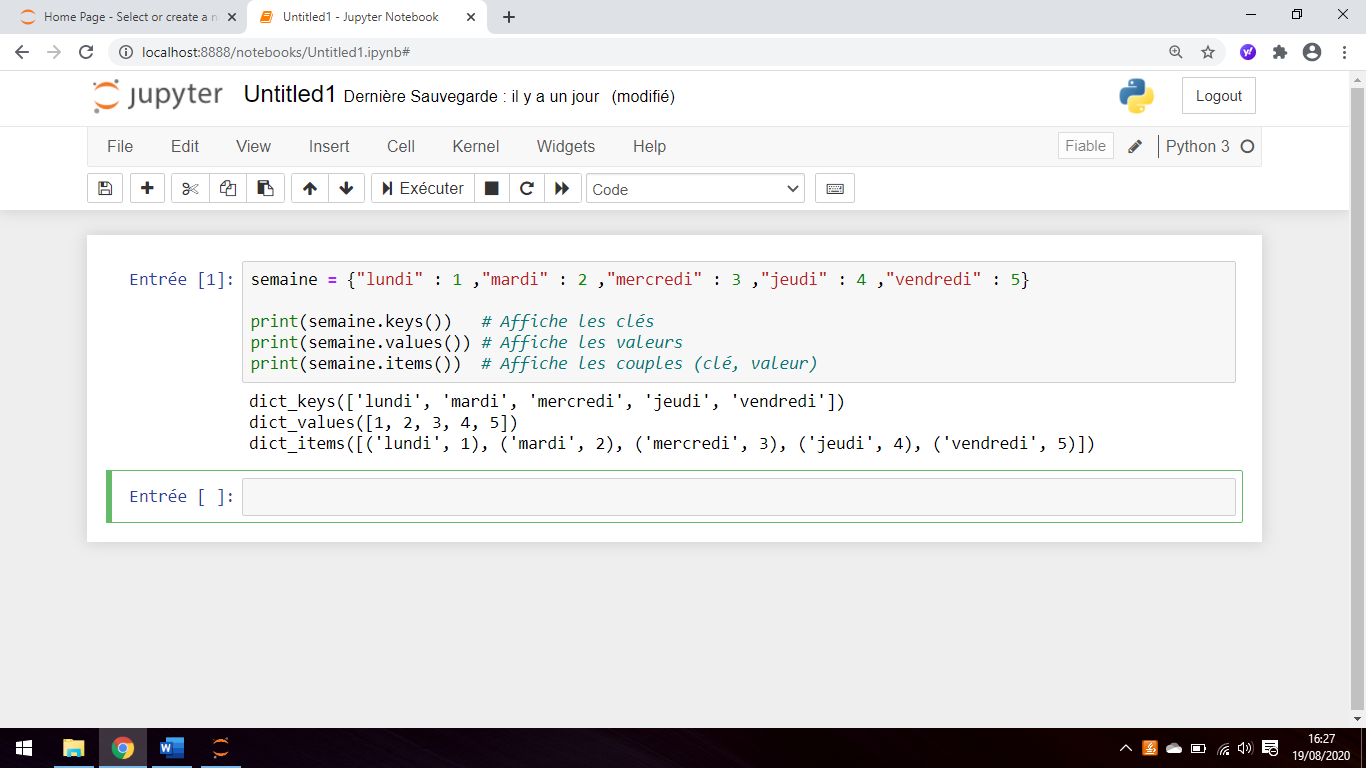


On notera la **syntaxe** avec les accolades qui délimitent le dictionnaire et les deux points séparant clé et valeur.

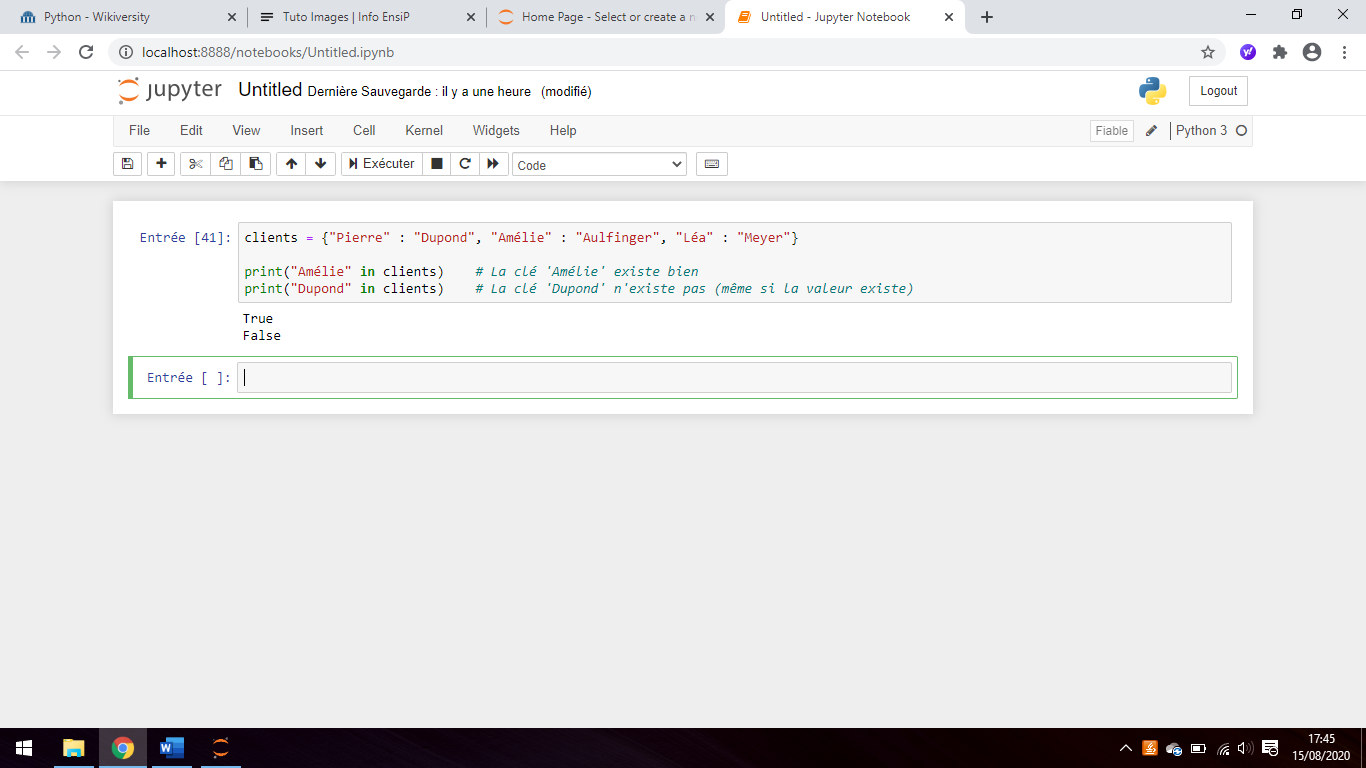
**2/ Accès aux éléments**

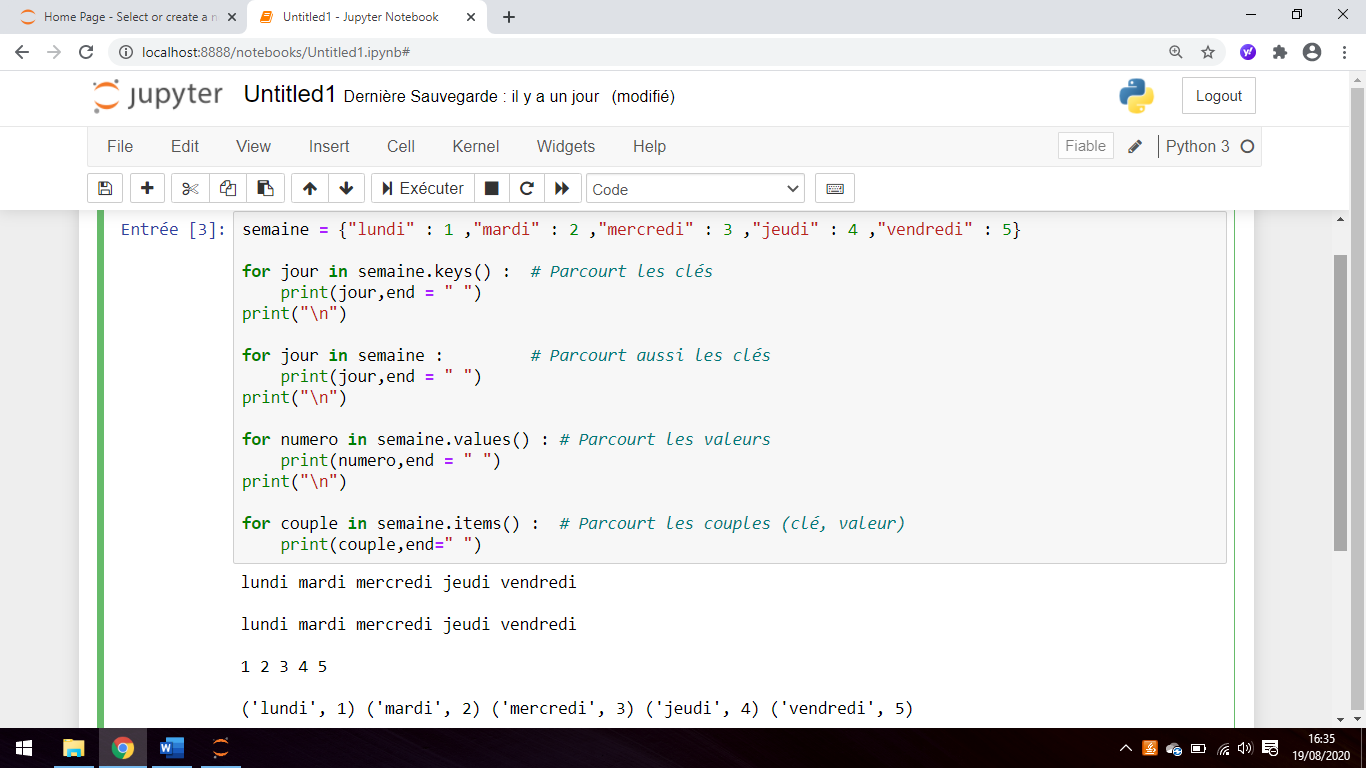
Un dictionnaire comprend des couples *(clé, valeur).* On peut accéder simplement soit aux couples, soit aux clés, soit aux valeurs.

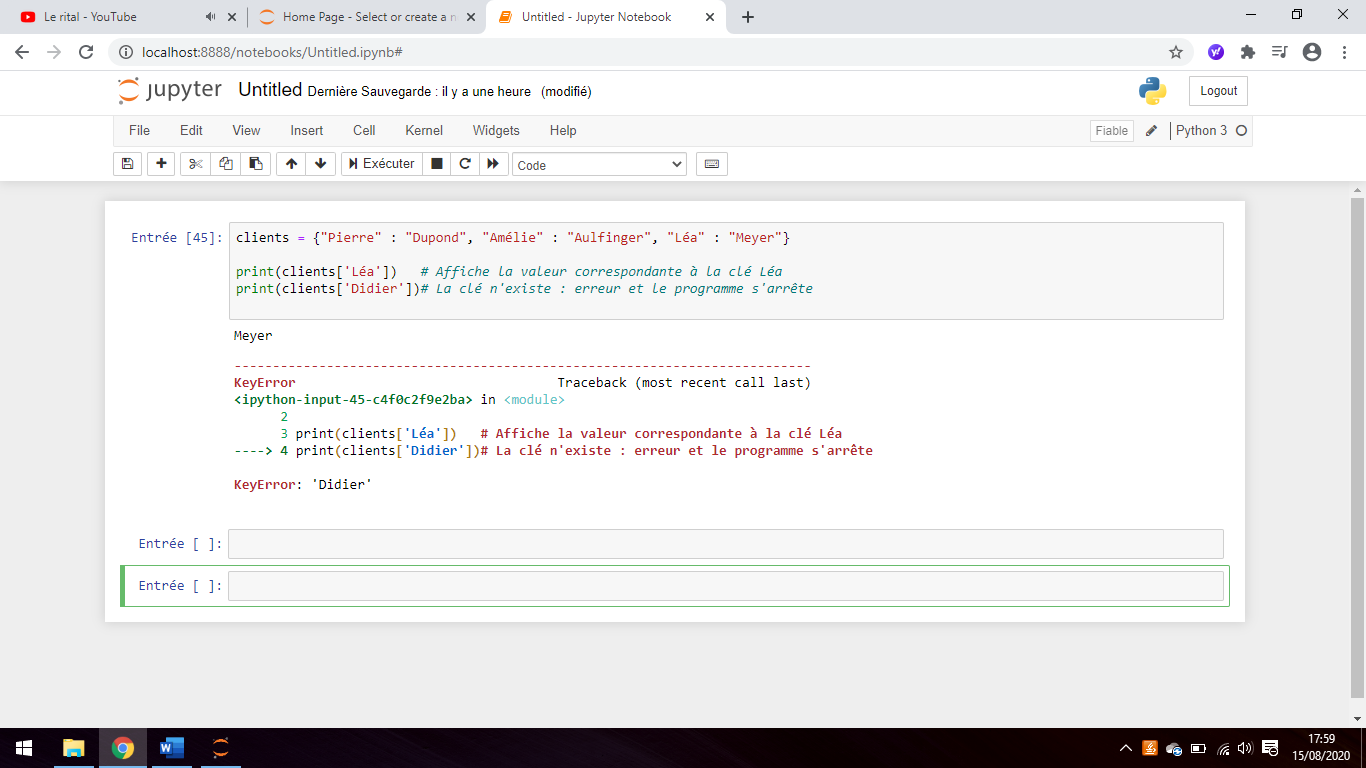
**A savoir** : la méthode **keys()** permet l’accès aux clés, la méthode **values()** celle aux valeurs et la méthode **items()** aux couples.



**A savoir** : Le mot clé **in** permet de savoir **si** **la clé testée existe bien** et non la valeur.

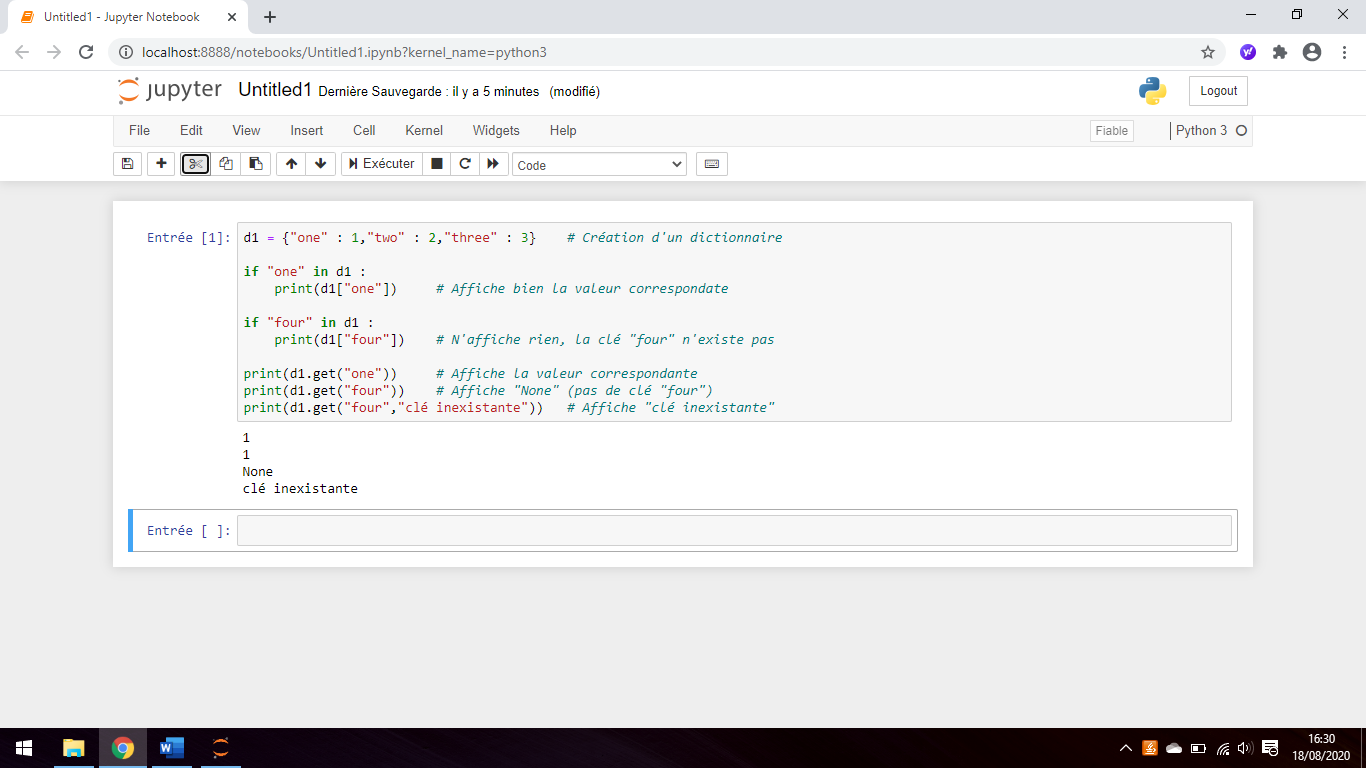


**Plusieurs méthodes pour parcourir un dictionnaire (à connaître)** :

**A savoir** : On peut simplement accéder à une valeur si l’on connait sa clé à l’aide de l’opérateur **[*clé*]** (comme pour les listes sauf que l’on donne une clé plutôt qu’un indice :

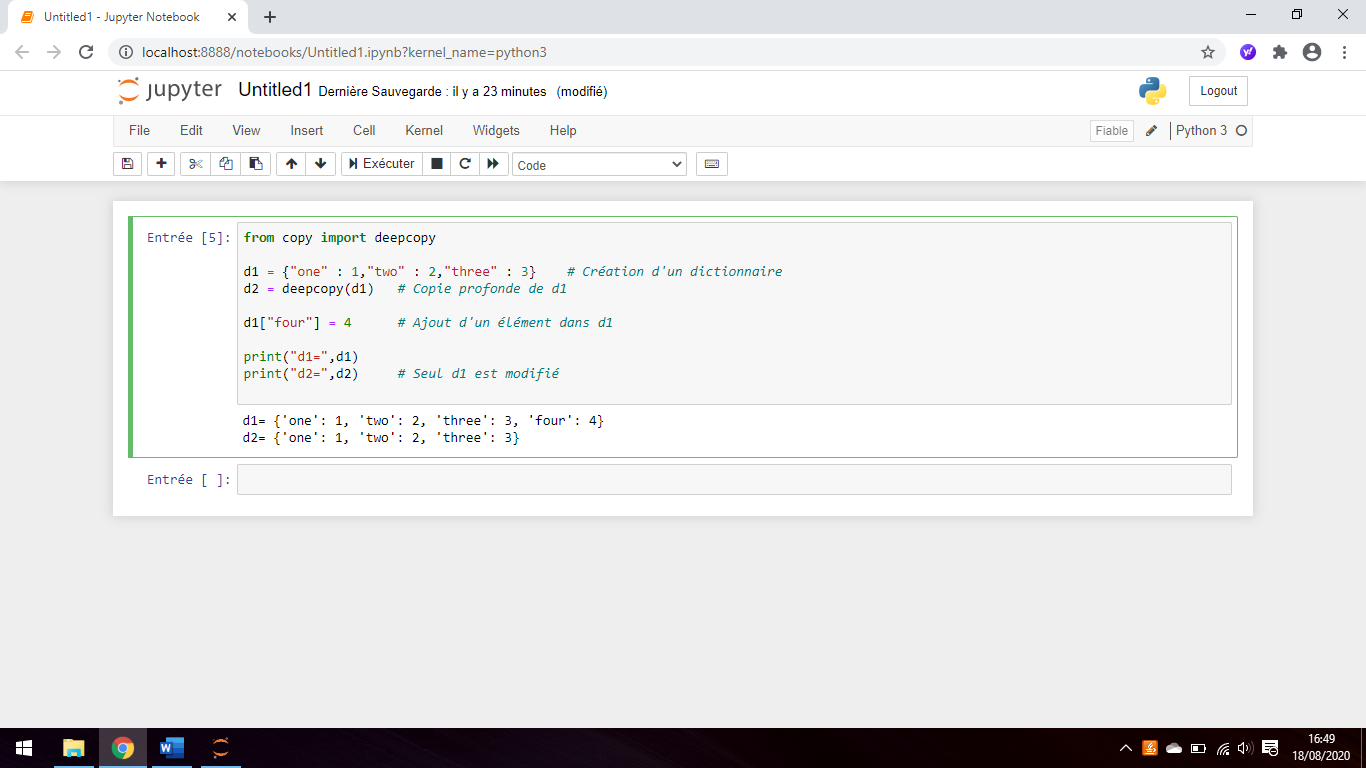
On remarque là aussi que s’il n’y a aucune valeur correspondante, le programme génère une erreur et s’arrête. Pour éviter cela, **deux méthodes possibles** (qui sont aussi **à connaître** mais vous avez l’habitude 😊 )

* Vérifier d’abord si la clé existe avec le mot clé **in.**
* Utiliser la méthode **get(*clé*)** du type *dict*.



**3/ Copie de dictionnaire**

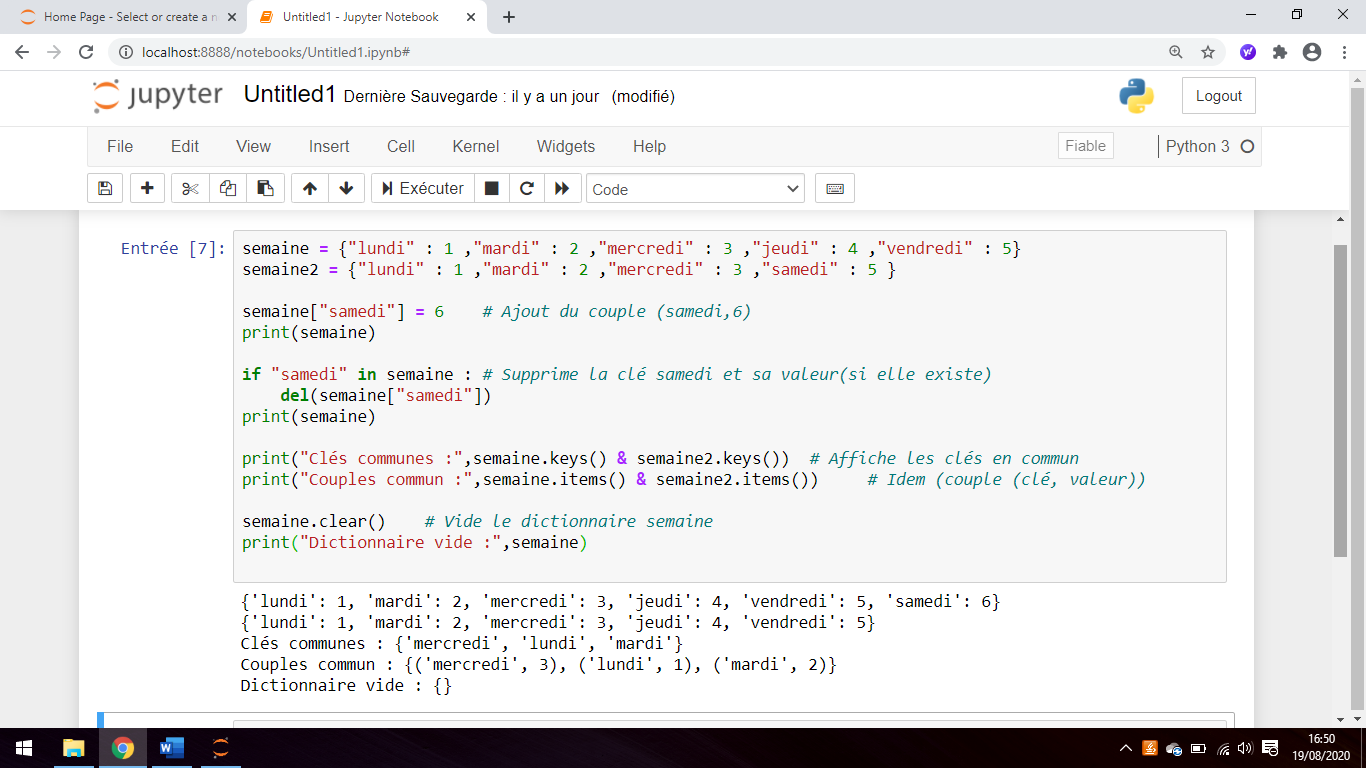
On retrouve les mêmes problèmes de références partagées qu’avec les listes: on utilisera là encore la fonction *deepcopy()* si l’on a besoin de copier un dictionnaire.



**4/ Autres méthodes utiles**

Voici quelques méthodes utiles avec un objet type *dict*. La liste n’est bien sûr pas exhaustive.

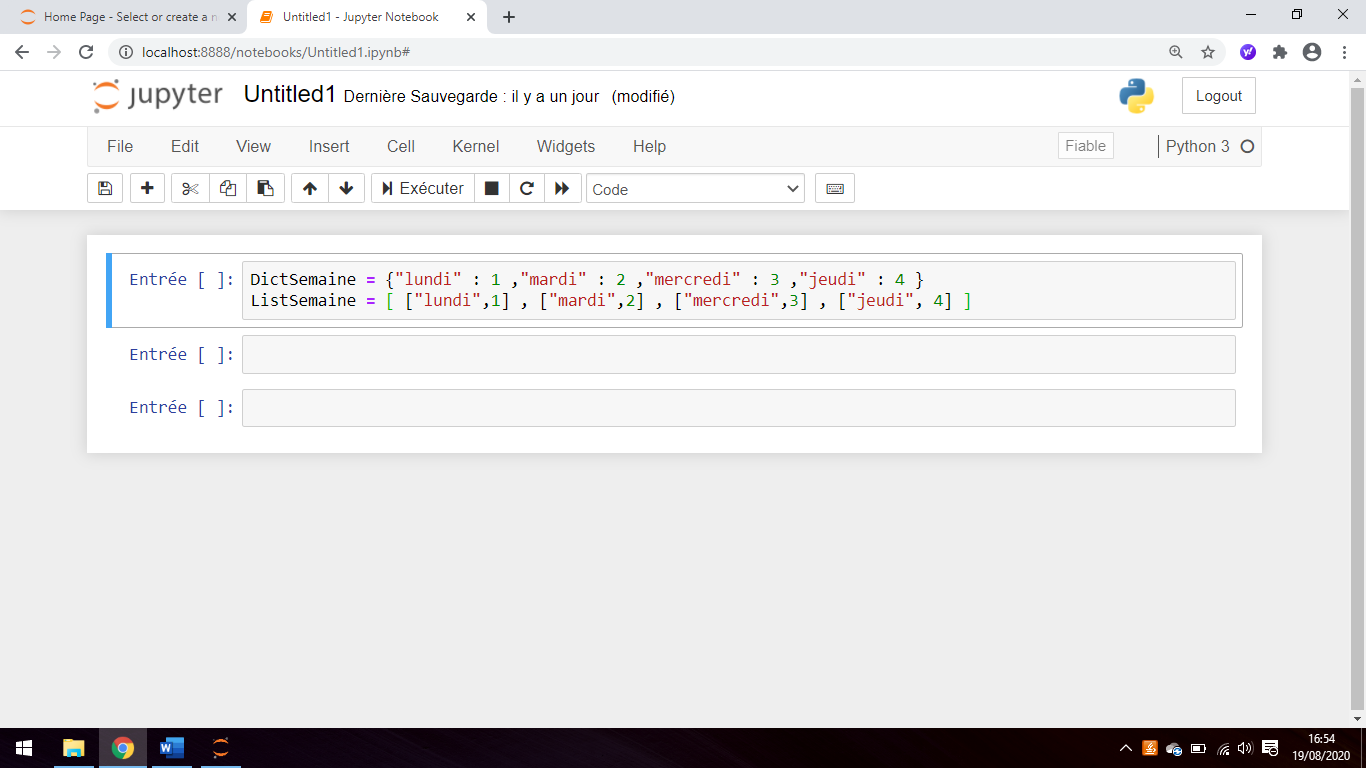
* L’expression **dict[clé] = valeur** ajoute le couple (clé, valeur) au dictionnaire.
* La fonction ***del*(dict[clé]))** efface un élément du dictionnaire.
* La méthode ***clear*()** vide le dictionnaire.
* L’opérateur **&** permet d’obtenir des éléments communs à deux dictionnaires.



**5/ Listes ou dictionnaires ?**

Il est vrai que l’on peut remplacer un objet *dict* comme une liste de listes à deux éléments.

Exemple :



Il faut toutefois bien saisir les différences entre les objets *list* et *dict* notamment le fait qu’une **liste est mutable** mais **pas** les **clés** d’un **dictionnaire**.

|  |  |
| --- | --- |
| **Avantage au dictionnaire** | **Avantage à la liste de listes** |
| * Peu de modifications de données. * Une donnée non modifiable. * Unicité d’une donnée. * Beaucoup de recherches d’éléments (dans un fichier texte par exemple). * Peu / pas d’ajouts d’éléments. | * Beaucoup de modifications de données. * Peu de recherches d’éléments en particulier. * De nombreux parcours de groupes d’éléments ou de l’objet en entier. * Plusieurs données identiques. * Pas de nécessité de trier systématiquement. |

**A noter** : cette notion de **« donnée non modifiable et unique »** est primordiale et largement utilisée en informatique (par exemple : date de naissance, numéro de sécurité sociale, numéro de facture etc.) et permet de sécuriser les échanges : c’est le principe des bases de données qui seront étudiées en Terminale.

**A noter** : les éléments d’un objet *dict* sont systématiquement triés dès qu’il est modifié : cela prend beaucoup de temps mais en contrepartie, toute recherche d’élément en particulier est bien plus efficace que dans une liste.