CHAPITRE 3 : Types construits

CHAPITRE 3 : Types construits

1 p-uplets, p-uplets nommés

1.1 Histoire des sciences : le langage Python en bref

- Le langage Python a été imaginé et développé par Guido Van Rossum en 1991.
- Le langage Python offre la possibilité de programmer selon différents paradigmes[1] et sur diverses plateformes[2]. Il est devenu l'un des langages les plus populaires, grâce notamment à sa facilité de prise en main. Il est particulièrement utilisé en intelligence artificielle grâce à de nombreux modules en évolution constante.
- Le langage Python utilise différents types de variables.
- Les types qui contiennent plusieurs éléments sont appelés types construits.

Parmi les types construits, on trouve :

- Le p-uplet (appelée *tuple* en anglais). Exemple `(2, 3, 6, 8)`
- La liste (appelée *list* en anglais). Exemple `[2, 3, 6, 8]`
- Le dictionnaire (appelé *dictionary* en anglais). Exemple `{2: 3, 6: 8}`

1.2 Découverte du p-uplet en Python

- Le p-uplet (souvent appelé *tuple* en Python) peut se décliner en 2-uplet ou doublet (en prenant p = 2), 3-uplet ou triplet (p = 3), 4-uplet ou quadruplet (p = 4), etc.

Exemples :

`(2, 3, 6, 8)` est un 4-uplet ou quadruplet. `(2, 4)` est un 2-uplet ou doublet.

1.3 Création d'un p-uplet : on utilise une affectation.

Pour créer un p-uplet il suffit d'écrire son nom, le signe d'affectation et sa valeur.

Exemple :

```
t1 = ('f', 'a', 'c', 'e')
```

[1] **Paradigme :** façon d'approcher la programmation. par exemple on peut voir le programme comme une collection d'objets (boutons, fenêtres, …). Un programme peut être aussi vu comme une suite d'évaluations de fonctions etc.

[2] **Plateforme :** c'est le système d'exploitation (Linux, Windows etc.)

1.4 Concaténation de deux p-uplets

Concaténer signifie mettre bout à bout.

On peut utiliser l'addition des deux p-uplets qu'on veut concaténer ou la multiplication d'un entier par le p-uplet qu'on veut concaténer plusieurs fois avec lui-même.

- Concaténation de deux p-uplets : on utilise le 'plus'.

Exemple :

Si les p-uplets `t1 et t2` valent respectivement

`('f', 'a', 'c', 'e')` et `('b', 'o', 'o', 'k')` alors leur concaténation est `t1 + t2` vaut `('f', 'a', 'c', 'e', 'b', 'o', 'o', 'k').`

- Concaténation d'un p-uplet plusieurs fois avec lui-même : on utilise le 'multiplié'.

Exemple :

Si le p-uplets `t1` vaut `('f', 'a', 'c', 'e')` alors sa concaténation avec lui-même 2 * `t1` vaut

`('f', 'a', 'c', 'e', 'f', 'a', 'c', 'e').`

1.5 Opérateur d'appartenance

- L'opérateur d'appartenance `in` permet de savoir si un élément est présent.

Exemples :

`'f' in t1` a pour valeur True.

`'b' in t1` a pour valeur False.

1.6 Index d'un élément

- Index ou indice d'un élément

Le 1er élément du tuple `t1` s'obtient par `t1[0].` Il vaut `'f'.`

Le 2e élément du tuple `t1` s'obtient par `t1[1].` Il vaut `'a'.`

Les index d'un quadruplet vont de 0 à 3.

1.7 Objet immutable

- Un p-uplet n'est plus modifiable par affectation une fois créé.

Si on essaye d'affecter une valeur à un élément déjà présent on obtient une erreur d'affectation.

Exemple :

`t1[2] = 's'` provoque une erreur. Il n'est pas possible de modifier un tuple après sa création. On dit qu'il n'est **pas muable** ou **pas mutable** ou encore **immutable**.

On peut contourner cela en créant un nouveau p-uplet et en copiant une partie de ses éléments.

Par exemple si on veut obtenir le p-uplet `('f', 'a', 's', 'e')` à partir des éléments de t1 alors on peut concaténer les éléments de t1 sauf celui qu'on veut modifier.

```
t1 = ('f', 'a', 'c', 'e')
t1 = (t1[0],) + (t1[1],) + ('s',) + (t1[3],)
```

`t1` a pour valeur `('f', 'a', 's', 'e')`.

1.8 Affection multiple ou dépaquetage

- **L'affectation multiple**, encore appelé le **dépaquetage** (*unpacking*) de p-uplet consiste à affecter à plusieurs variables toutes les valeurs des éléments d'un p-uplet avec un seul symbole d'affectation =.

Remarques :

L'instruction Python `a, b, c, = (10, 15, 20)` ne pourra fonctionner que si 3 variables `a, b, c` sont utilisées.

Cela revient à faire parallèlement les trois affectations

`a = 10` `b = 15` `c = 20.`

Des instructions du type `a, b = (10, 15, 20)` ou encore `a, b, c, d = (10, 15, 20)` provoquent une erreur, car le p-uplet `(10, 15, 20)` contient 3 valeurs exactement, mais pas 2 ni 4.

Le dépaquetage de p-uplet apparaît souvent dans les QCM des épreuves communes et sera réintroduit (et donc nécessaire) dans l'implémentation d'algorithmes plus complexes.

Exemple :

- Soit le programme :

```
liste_eleves = [('Jean', 'Dugarden', 2005, 'TG1'), ('Tom', 'Crouse', 2002, 'TG2')]

def recherche(nom, liste_eleves):
    for eleve in liste_eleves:
        if eleve[1] == nom:
            return eleve
    return None, None, None, None
```

```
(a, b, c, d) = recherche('Dugarden', liste_eleves)
print((a, b, c, d))

(a, b, c, d) = recherche('Leblanc', liste_eleves)
print((a, b, c, d))
```

- En l'exécutant on a :

```
('Jean', 'Dugarden', 2005, 'TG1')

(None, None, None, None)
```

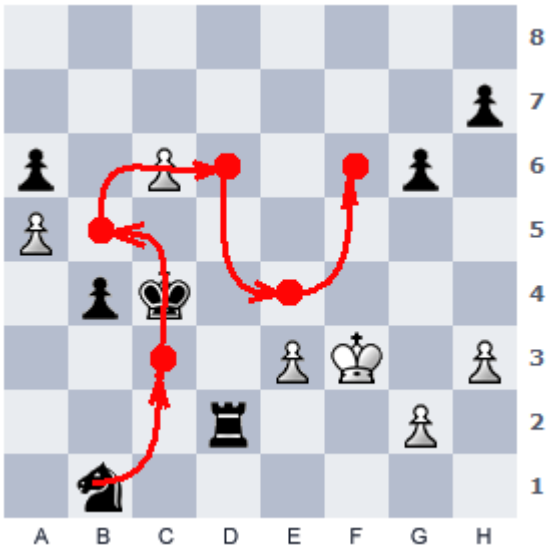
Il est important d' observer que le comportement de la fonction est identique, qu'un élève soit trouvé ou non.

- Dans le cas où un élève a été trouvé, le quadruplet (a, b, c, d) est renvoyé.
- Si aucun élève n'a été trouvé, la fonction renvoie le quadruplet (None, None, None, None).

La fonction doit renvoyer un quadruplet dans tous les cas afin de ne pas créer d'erreur de dépaquetage.

2 Tableau indexé, tableau donné en compréhension

- Les tableaux sont des structures de données informatiques. En Python, on utilise les listes pour les écrire.
- On peut par exemple stocker les six positions successives du cheval d'un jeu d'échecs dans une liste de listes.



Il suffit de créer une liste de six sous-listes.

Chacune des six sous-listes contient deux éléments qui sont l'abscisse et l'ordonnée des positions sur l'échiquier.

```
L = [['B', 1], ['C', 3], ['B', 5], ['D', 6], ['E', 4], ['F', 6]]
```

2.1 Découverte du type liste

Un autre type de variable, très utilisé en Python, est appelé « liste ». La liste en Python peut contenir des objets de différents types.

Exemples :

`L1 = [0, 1, 2, 3, 4]` contient cinq **nombres entiers**.

`L2 = ['1', '2', '3']` contient cinq **caractères**.

Une liste en Python représente ce qu'on appelle de façon très générale un "tableau".

On pourrait par exemple représenter L1 ainsi :

0	1	2	3	4

Il est aussi possible d'imbriquer différents types de variables, telles des poupées russes, pour obtenir, par exemple, une liste de listes ou encore une liste de p-uplets, etc.

Exemple : `L = [1, 2.8, '3', True, ('a', 'b', 'c')]` contient cinq **éléments de différents types** (un nombre entier, un nombre flottant, un caractère, un booléen, un p-uplet).

2.2 Création d'une liste : on peut utiliser une affectation.

Exemple : `t1 = ['f', 'a', 'c', 'e']`

2.3 Création d'une liste : on peut utiliser une liste en compréhension *expression* for x in *valeurs de départ*

Cette syntaxe très particulière est particulièrement efficace et doit être bien assimilée.

[*expression* for x in *valeurs de départ*]

Exemple 1

Que vaut `L = [x**2 for x in valeurs_de_depart]` avec :

$$\begin{cases} expression = x ** 2 \\ valeurs\ de\ départ = [2, 3, 4] \end{cases} ?$$

Réponse

`L` vaut `[4, 9, 16]`

Exemple 2 Que vaut `L3 = [j**2 for j in range(5)]` ?

Réponse

On a : $\begin{cases} expression = j ** 2 \\ valeurs\ de\ départ = [0,1,2,3,4] \end{cases}$

donc

`L3` vaut `[0, 1, 4, 9, 16]`

Exemple 3 Donnez la valeur de la liste L4 définie en compréhension par :

```
import math
L4 = [math.sqrt(element) for element in L3]
```

Réponse

On a : $\begin{cases} expression = \sqrt{element} \\ valeurs\ de\ départ = [0,1,4,9,16] \end{cases}$

donc

`L4` vaut `[0.0, 1.0, 2.0, 3.0, 4.0]`

2.4 Ajout d'une *condition de filtrage* à la fin d'une liste en compréhension

[***expression*** `for` x `in` ***valeurs de départ*** `if` ***booléen de filtrage***]

Exemple

Donnez la valeur de la liste L définie par : `[j**2 for j in range(5) if (j**2)%2 == 0]`

Réponse

`j**2 for j in range(5)` donne `[0, 1, 4, 9, 16]`, mais le booléen de filtrage est vrai lorsque le reste de la division de `j**2` par 2 est nul autrement dit lorsque `j**2` est pair donc

L vaut `[0, 4, 16]`

2.5 Création d'une liste de listes

Dans une liste on peut mettre d'autres listes. On peut utiliser :

- Une boucle for contenue dans une autre boucle for (exemple 1).
- Une liste en compréhension contenue dans une autre liste en compréhension (exemple 2).

Exemple 1: Que vaut la liste de listes M après l'exécution du programme :

```
M = []
for i in range(3):
    ligne = []
    for j in range(5):
        ligne.append(i*j)
    M.append(ligne)
```

Réponse : M vaut `[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8]]`.

Dans l'exemple suivant, on crée la même liste de listes M. Mais on utilise deux listes en compréhension.
On commence par écrire la boucle `for j` la plus interne. Elle 'fabrique' une ligne.

Exemple 2:

Nombre de colonnes

Nombre le lignes

```
M =[[i*j for j in  range(5)] for i in  range(3)]
```

expression

avec $\begin{cases} expression = [\ ligne\ définie\ en\ compréhension] \\ valeurs\ de\ départ = [0, 1, 2] \end{cases}$

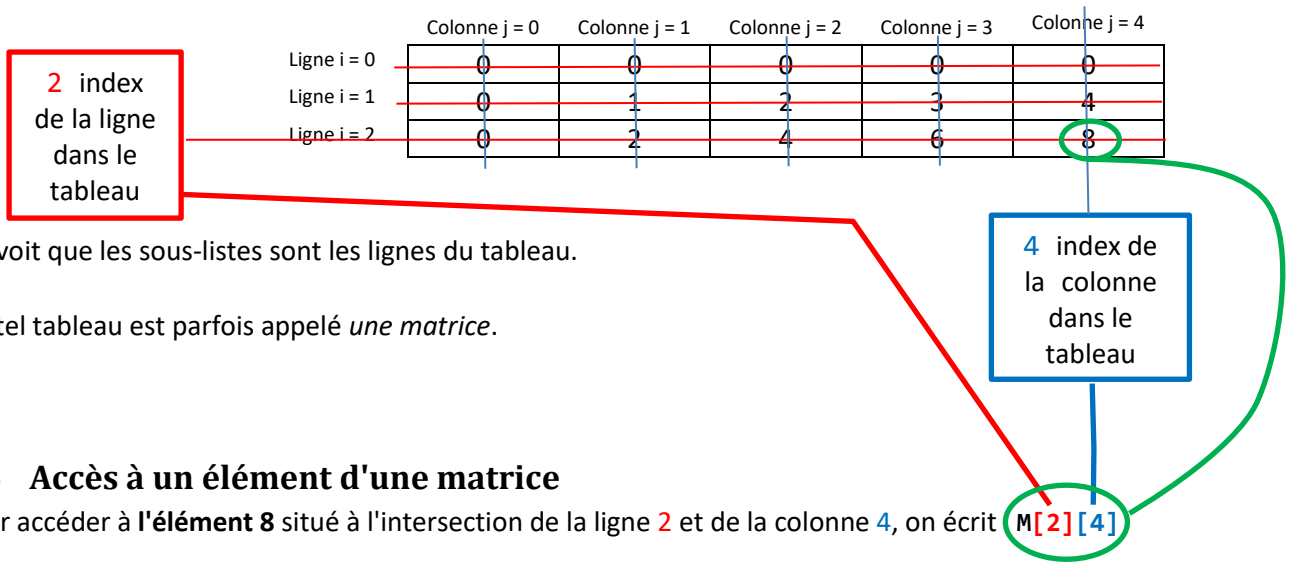
M contient donc 3 lignes contenant chacune 5 colonnes :

`[[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8]]`

`i` vaut `0` — `i` vaut `1` — `i` vaut `2`

- Chaque sous-liste est une "ligne".
- La première sous-liste est calculée en prenant en compte `i = 0` et les 5 valeurs de `j`.
- La deuxième sous-liste est calculée en prenant en compte `i = 1` et les 5 valeurs de `j`. etc.

On peut représenter la listes de listes M sous forme de tableau :



On voit que les sous-listes sont les lignes du tableau.

Un tel tableau est parfois appelé *une matrice*.

2.6 Accès à un élément d'une matrice

Pour accéder à **l'élément 8** situé à l'intersection de la ligne 2 et de la colonne 4, on écrit `M[2][4]`

```
M = [[0, 0, 0, 0, 0], [0, 1, 2, 3, 4], [0, 2, 4, 6, 8]]
```

```
M[n° de ligne][n° de colonne]
```

`M[2][4]` vaut `8`

⚠ Pour désigner un élément, mettre l'index de la ligne en premier … et il vaut 0 *en haut*.

2.7 Méthode pour ajouter un élément à une liste

Pour ajouter un élément **à la fin** d'une liste L **déjà créée**, on utilise la méthode **append()**.

Exemple :

- On crée la liste `ma_liste` qui vaut `[2, 35, 42, 39]`.
- On exécute l'instruction `ma_liste.append(41)`.

`ma_liste` vaut alors `[2, 35, 42, 39, 41]`.

2.8 Accéder à un élément d'une liste par son index

On peut accéder individuellement à un élément d'une liste grâce à son index (ou indice) qui est sa position dans la liste. Les index débutent toujours à zéro en Python.

Exemple : Soit la liste `fruits = ['pomme', 'banane', 'raisin', 'orange', 'pêche']`.

`fruits[0]` a pour valeur `'pomme'`.

`fruits[1]` a pour valeur `'banane'`.

`fruits[2]` a pour valeur `'raisin'`.

`fruits[3]` a pour valeur `'orange'`.

`fruits[4]` a pour valeur `'pêche'`.

Mais aussi, on peut utiliser les index négatifs qui commencent par la fin et qui remontent au début :

`fruits[-1]` a pour valeur `'pêche'`.

`fruits[-2]` a pour valeur `'orange'`.

`fruits[-3]` a pour valeur `'raisin'`.

`fruits[-4]` a pour valeur `'banane'`.

`fruits[-5]` a pour valeur `'pomme'`.

Conclusion :

Dans une liste de longueur 5, on peut utiliser les valeurs d'index de -5 à 4.

2.9 Opérateur d'appartenance

- L'opérateur d'appartenance `in` permet de savoir si un élément est présent dans une liste.

Exemples :

`'pomme' in fruits` a pour valeur True.

`'cassis' in fruits` a pour valeur False.

2.10 Longueur d'une liste

En anglais, longueur se dit *length*. La fonction `len()`, commune aux trois types construits, qui renvoie le nombre d'éléments de la liste.

Exemple :

`len(fruits)` vaut 5.

2.11 Comptage d'occurrences d'un élément donné dans une liste donnée

On utilise une variable `compteur`, gérée par une variable de type `int`.

Exemple :

```
fruits = ['pomme', 'banane', 'raisin', 'orange', 'pêche']


def compte(L, element):
    compteur = 0
    for x in L:
        if x == element:
            compteur = compteur+1
    return compteur
```

`print(compte(fruits, 'banane'))` affiche 1

En résumé, la fonction qui compte le nombre d'occurrence du mot 'banane' dans la liste fruits utilise un compteur initialisé à zéro.

La boucle for parcourt la liste et si un élément vaut 'banane' alors le compteur est incrémenté de 1.

2.12 Modifier une liste par affectation

On peut modifier un élément d'une liste en lui affectant une nouvelle valeur.

Rappel : c'est impossible avec un p-uplet.

Exemple :

```
fruits[1] = 'ananas'
```

Si on affiche la liste fruits on obtient :

```
['pomme', 'ananas', 'raisin', 'orange', 'pêche']
```

On dit qu'une liste est **muable**, **modifiable** ou encore **mutable**.

Au contraire, un p-uplet est **immuable**, **non modifiable** ou encore **non mutable**.

2.13 La méthode append() pour ajouter un élément à une liste

La méthode `append()`opère uniquement sur le type `list`. Elle permet d'ajouter un élément à la fin d'une liste en allongeant cette liste d'un élément.

La syntaxe **point**, empruntée à la programmation orientée objet (POO) est ici nécessaire. Ainsi l'instruction Python `L.append(2)` ajoute-t-elle l'entier 2 en dernière position de la liste `L`.

Notons bien que cet ajout dynamique a une complexité très défavorable : pour ajouter un élément à une liste, il faut d'abord créer une nouvelle variable de taille supérieure, ensuite recopier tous les éléments un par un, pour enfin copier en dernière position l'élément à ajouter. Tout ceci est transparent en Python, comme un genre de boîte noire.

La méthode doit être connue car beaucoup de questions des évaluations communes l'utilisent.

Exemple 1 :

Soir la liste L3 qui vaut `[0, 1, 4, 9, 16]`

Après l'exécution de l'instruction `L3.append(18)` la liste L3 a pour valeur `[0, 1, 4, 9, 16, 18]`

Exemple 2:

On peut construire une liste en partant d'une liste vide `L = []`

On écrit ensuite une boucle for qui contient l'instruction `L.append(expression)`

Ainsi à chaque tour de boucle un nouvel élément est ajouté à partir d'une expression.

```
def modification(liste_origine):
    liste_modifiee = []
    for x in liste_origine:
        liste_modifiee.append(x**2)
    return liste_modifiee

L = modification([1, 2, 3, 4, 5])
print(L)
```

affiche

```
[1, 4, 9, 16, 25]
```

2.14 Le "slicing" encore appelé "découpage" de liste

Le langage Python se prête très bien au *slicing*. Cet exercice propose d'en découvrir un usage simple en opérant uniquement avec les deux arguments `debut` et `fin`, selon la syntaxe `L[debut:fin]`. Il existe un autre argument possible appelé le pas selon la syntaxe `L[debut:fin:pas]`. Dans ce cas, les éléments sont extraits toutes les `pas` valeurs. Il est recommandé de se référer à la documentation Python pour découvrir encore plus de possibilités sur le *slicing*.

Attention, le *slicing* peut être l'objet de questions lors de l'évaluation commune de 1re.

Exemple :

```
mois = ['janvier', 'février', 'mars', 'avril', 'mai', 'juin', 'juillet', \
        'août', 'septembre', 'octobre', 'novembre', 'décembre']

extrait_1 = mois[0 : 6]
extrait_2 = mois[6 : 8]
print(extrait_1)
print(extrait_2)
```

affiche :

```
['janvier', 'février', 'mars', 'avril', 'mai', 'juin']
['juillet', 'août']
```

Remarques :

`mois[0 : 6]` contient les éléments depuis `mois[0]` **inclus** jusqu'à `mois[6]` **exclu**.

La présence de l'antislash (symbole \ ) dans l'écriture de la liste permet de passer à la ligne en cas d'écriture d'une liste trop longue dans un éditeur Python.

3 Dictionnaires par clés et valeurs

3.1 Création à partir d'un dictionnaire et écriture des entrées une à une

Voici un type nouveau de variable, le type dictionnaire. Contrairement au p-uplet et à la liste, *on n'accède pas à un élément par son index* de position.

On accède à un élément par sa **clé.** Les clés sont *toutes différentes* et de type **non** muable.

Une clé peut être un entier, une chaîne de caractères, un p-uplet. Il est conseillé d'utiliser des chaînes de caractères.

A une clé, on fait correspondre une **valeur** qui peut être un entier, une chaîne de caractères, un p-uplet, une liste, un booléen...

Exemple :
Construire à la main trois entrées d'un dictionnaire de mots. Il n'est pas obligatoire que ce soit un dictionnaire de mots mais cet exemple est choisi afin de rester dans la symbolique du mot *dictionnaire*.

```
dico = {}

dico['bit'] = "élément d'information valant 0 ou 1"

dico['processeur'] = "unité de calcul de l'ordinateur"

dico['RAM'] = "Mémoire vive"
```

Remarques :

- Dans cet exemple, toutes les **clés** 'bit', 'processeur', 'RAM' sont du type string (chaînes de caractères).
- Les **valeurs** associées à ces clés sont aussi du type string. Toutefois on a utilisé des doubles guillemets pour permettre les apostrophes comme dans `"élément d'information valant 0 ou 1".`
- Comme pour les p-uplets et les listes, la fonction `len()` permet de connaitre le nombre d'éléments.

3.2 Affichage d'un dictionnaire

Un dictionnaire se présente de la façon suivante :

```
{'clé1': "valeur1", 'clé2': "valeur2", 'clé3': "valeur3" }
```

Exemple :

Si on affiche le dictionnaire créé dans l'exemple du paragraphe précédent alors on obtient :

```
{'bit': "élément d'information valant 0 ou 1", 'processeur': "unité de calcul de l'ordinateur", 'RAM': "Mémoire vive"}
```

On voit qu'une clé et sa valeur correspondante sont séparées par deux points et que les couples **clé: valeur** sont séparés par des virgules. Les couples **clé: valeur** sont appelés les "entrées" ou **items** du dictionnaire.

3.3 Création d'un dictionnaire en écrivant de l'ensemble des items

Exemple :

```
mots = {'Avoir': 'have, had, had', 'Savoir': 'know, knew, known',\
        'Aller': 'go, went, gone', 'Prendre': 'take, took, taken',\
        'Chercher': 'seek, sought, sought'}
```

Noter encore la présence des antislash dans le dictionnaire pour couper les lignes trop longues.

3.4 Suppression d'une entrée d'un dictionnaire par l'instruction del dico[clé]

Exemple :

`del mots['Avoir']` supprime l'item `'Avoir': 'have, had, had'` du dictionnaire `mots`.

⚠ Si la clé est absente du dictionnaire, alors on obtient une erreur `KeyError`.

3.5 Méthodes spécifiques aux dictionnaires : keys(), values() et items()

Ces méthodes doivent être bien connues car de nombreuses questions des évaluations communes y font référence.

Comme pour les p-uplets et les listes, l'opérateur `in` permet de faire un test d'appartenance.

On rappelle que Python est sensible à la casse, c'est-à-dire qu'une même lettre ne signifie pas la même chose selon qu'elle est en majuscule ou en minuscule.

- `mon_dico.keys()` permet d'extraire toutes les clés du dictionnaire `mon_dico`.
- `mon_dico.values()` permet d'extraire toutes les valeurs du dictionnaire `mon_dico`.
- `mon_dico.items()` permet d'extraire toutes les entrées du dictionnaire `mon_dico`.

Exemple :

Si on a le dictionnaire

```
mots = {'Avoir': 'have, had, had', 'Savoir': 'know, knew, known',\
        'Aller': 'go, went, gone', 'Prendre': 'take, took, taken',\
        'Chercher': 'seek, sought, sought'}
```

alors :

Instruction	Valeur renvoyée
`mots.keys()`	`dict_keys(['Avoir', 'Savoir', 'Aller', 'Prendre', 'Chercher'])`
`mots.values()`	`dict_values(['have, had, had', 'know, knew, known', 'go, went, gone', 'take, took, taken', 'seek, sought, sought'])`
`mots.items()`	`dict_items([('Avoir', 'have, had, had'), ('Savoir', 'know, knew, known'), ('Aller', 'go, went, gone'), ('Prendre', 'take, took, taken'), ('Chercher', 'seek, sought, sought')])`
`list(mots.keys())` ou `list(mots.values())` ou `list(mots.items())` renvoient les listes seules.	

3.6 Fonction pour concaténer les valeurs et les clés

```
def concatener(mots):
    c = ""
    for valeur in mots.values():
        c = c + valeur + ", "
    return c
print(concatener(mots))
```

affiche :

```
have, had, had, know, knew, known, go, went, gone, take, took, taken, seek, sought
, sought,
```

3.7 Ajout d'une nouvelle entrée dans un dictionnaire

Pour ajouter une entrée (c'est-à-dire un item) à un dictionnaire qui existe déjà, il suffit d'affecter une valeur **à une nouvelle clé** qu'on est libre de choisir.

Exemple :

```
mots = {'Avoir': 'have, had, had', 'Savoir': 'know, knew, known',\
        'Aller': 'go, went, gone', 'Prendre': 'take, took, taken',\
        'Chercher': 'seek, sought, sought'}
```

```
mots['Etre'] = 'be, was, been'
```

On peut ensuite vérifier que le dictionnaire mots a pour valeur :

```
{'Avoir': 'have, had, had', 'Savoir': 'know, knew, known', 'Aller': 'go, went,
 gone', 'Prendre': 'take, took, taken', 'Chercher':
'seek, sought, sought', 'Etre': 'be, was, been'}
```

 Si on affecte une valeur **à une clé qui existe déjà**, l'ancienne valeur et écrasée par la nouvelle sans qu'il y ait d'alerte !

Exemple :

```
mots = {'Avoir': 'have, had, had', 'Savoir': 'know, knew, known',\
        'Aller': 'go, went, gone', 'Prendre': 'take, took, taken',\
        'Chercher': 'seek, sought, sought'}
mots['Avoir'] = 'bonjour'
```

On peut ensuite vérifier que le dictionnaire mots a pour valeur :

```
{'Avoir': 'bonjour', 'Savoir': 'know, knew, known', 'Aller': 'go, went, gone',
 'Prendre': 'take, took, taken', 'Chercher': 'seek, sought, sought'}
```

3.8 Opérateur d'appartenance

- De même que pour les p-uplets et les listes, l'opérateur d'appartenance `in` permet de savoir si un élément est présent dans un dictionnaire.

Exemple : Pour rechercher si une clé existe dans un dictionnaire on peut créer la fonction :

```
def cherche(verbe, mots):
    return verbe in mots.keys()
```

Elle renvoie `True` si l'argument `verbe` est présent dans les clés du dictionnaire `mots` et `elle` renvoie `False` sinon.

3.9 Les différents types de variables

Types de base	
`int`	Nombre entier
`float`	Nombre flottant
`str`	Chaîne de caractères
`bool`	Booléen

Types construits	
`tuple`	p-uplet
`list`	Liste
`dict`	Dictionnaire