

## Types de séquences

Des collections ordonnées d'objets: tuple, list, dict

a	type(a)
a = (1, 2, 3)	<class 'tuple'>
a = [1, 2, 3]	<class 'list'>
a = {1 : 2, 3 : 4}	<class 'dict'>

## Tuple

Un tuple est une séquence *non mutable*. Le tuple partage de nombreuses caractéristiques des listes, mais il n'est pas possible de le modifier (*non mutable*):

```
1. L = [1, 2, 3]
2. L[0] = 2
3. print(L)
4. # affiche . . .
5. T = (1, 2, 3)
6. print(T[0])
7. # affiche . . .
8. T[0] = 2
9. # affiche TypeError: . . .
```

**Usages:** le tuple peut représenter ...

- un point à partir de ses coordonnées (x, y)
- une date (jour, mois, année)
- une carte (10, 'coeur')
- ...

Une fonction peut **retourner un tuple**: Remarquer qu'il n'est pas nécessaire de mettre des parenthèses après le return.

```
1. def translation(a,b):
2.     """retourne un tuple constitué des positions x et y """
3.     x = a + 10
4.     y = b
5.     return x, y
6.
7. translation(5, 3)
8. # affiche . . .
```

On peut alors utiliser une **affectation multiple** pour stocker chacune des valeurs dans une variable:

```
1. z, k = translation(5, 3)
2. print(z, k)
3. # affiche . . .
```

## Dictionnaire

Un dictionnaire est une table associative qui fait correspondre des clés à des valeurs:

```
dic1 = {clé1: valeur1, clé2: valeur2, clé3: valeur3}
```

L'ordre des couples n'est pas important: on peut aussi écrire:

```
dic1 = {clé3: valeur3, clé1: valeur1, clé2: valeur2}
```

*Exemples:*

```
liens = {1 : (2,3), 2: (1,3), 3: (1,2)}
```

```
capitales = {'France': 'Paris', 'Italie': 'Rome', 'Allemagne': 'Berlin'}
```

```
personnes = {('Lovelace', 'Ada'): (10, 'decembre', 1815, 'Londres'),
              ('Von Neumann', 'John'): (28, 'decembre', 1903, 'Budapest')}
```

instruction	commentaire
<code>liens[1] = (2,3,4)</code>	
<code>capitales['Angleterre'] = 'Londres'</code>	
<code>personnes[('Turing', 'Alan')] = (23, 'juin', 1912, 'Londres')</code>	

Tester la présence d'une clé avec le mot clé `in`:

instruction	sortie
<code>'France' in capitales</code>	True
<code>'Belgique' in capitales</code>	False

## Enumérer les clés

- avec la même méthode que pour les listes: `in`

```
1. for p in capitales:
2.     print(p)
3. # affiche . . .
```

- avec la liste des clés: `keys()`

```
1. pays = capitales.keys()
2. for p in pays:
3.     print(p)
4. # affiche . . .
```

---

## Enumérer les valeurs: `values()`

```
1. for val in capitales.values():
2.     print(val)
3. # affiche . . .
```

---

## Enumérer les paires clé/valeur `items()`

```
1. for cle, val in capitales.items():
2.     print(cle, val)
3. # affiche . . .
```

---

## Dictionnaires par compréhension

Le principe est le même que pour les listes.


```
1. dico = {x: 2 * x for x in range(4)}
2. print(dico)
3. # affiche
4. {0: 0, 1: 2, 2: 4, 3: 6}
```

---

## Construire un dictionnaire à partir d'un fichier csv

On utilise la méthode par compréhension. On importe le contenu du fichier csv dans l'objet **fichier**. Le dictionnaire est construit en itérant l'objet **reader** et en accédant à ses deux premières colonnes comme la paire clé-valeur du dictionnaire.

```
1. import csv
2. dict_from_csv = {}
3.
4. with open('csv_file.csv', mode='r') as fichier:
5.     reader = csv.reader(fichier)
6.     dict_from_csv = {rows[0]:rows[1] for rows in reader}
7.
8. print(dict_from_csv)
9. # affiche
10. {'France': 'Paris', 'Italie': 'Rome', 'Allemagne': 'Berlin'}
```



```
France,Paris
Italie,Rome
Allemagne,Berlin
```

*csv\_file.csv*