

# Plan du cours

---

<b>Chapitre 1</b>	<b>Fondamentaux du langage Python</b>
<b>Chapitre 2</b>	<b>Conteneurs standards en Python</b>
<b>Chapitre 3</b>	<b>Fonctions &amp; Générateurs</b>
<b>Chapitre 4</b>	<b>Passage de programmation impérative au fonctionnelle</b>
<b>Chapitre 5</b>	<b>Gestion des fichiers</b>
<b>Chapitre 6</b>	<b>Gestion des exceptions</b>

## Chapitre 4

# Programmation Python

## Chapitre 4 : passage de la programmation impérative au fonctionnelle

- ❑ Introduction à la programmation fonctionnelle
- ❑ Fonction pure
- ❑ Fonction de première classe
- ❑ Fonction imbriquée et closures
- ❑ Fonction d'ordre supérieur
- ❑ Fonction anonyme `lambda`
- ❑ Fonctions natives `map`, `filter`, `zip`
- ❑ Fonction récursive
- ❑ Module `itertools`

# Définition et concepts clés

## ❑ Qu'est-ce que la programmation fonctionnelle ?

langage



framework

Paradigme de  
programmation



Une façon  
d'aborder la  
programmation 😊

## ❑ Pourquoi la programmation fonctionnelle ?

- L'accent est mis sur les **transformations de données** et sur **l'utilisation de fonctions sans effets de bord**.
- Bien que les boucles `for` et `while` soient moins couramment utilisées dans ce paradigme, elles ne sont pas interdites.
- Afin d'avoir un code plus robuste et plus stable, donc moins de bugs et moins de maintenances.

# Différences entre programmation fonctionnelle et programmation impérative.

```
# Filtrer les nombres pairs
def filtrer_pairs_imp(liste):
    pairs = []
    for nombre in liste:
        if nombre % 2 == 0:
            pairs.append(nombre)
    return pairs

numbers = [1,2,3,4,5,6]
result = filtrer_pairs_imp(numbers)
print(result) #Affiche [2, 4, 6]
```

## Programmation impérative

- Décrire explicitement les étapes que l'ordinateur doit suivre pour accomplir une tâche.
- Se concentre sur la modification de l'état du programme en utilisant des **instructions séquentielles** (avec des boucles..).

```
# Filtrer les nombres pairs
def filtrer_pairs_func(liste):
    return list(filter(lambda x: x % 2 == 0, liste))

nombres = [1, 2, 3, 4, 5, 6]
resultat = filtrer_pairs_func(nombres)
print(resultat) # Affiche [2, 4, 6]
```

## Programmation fonctionnelle

# Différences entre programmation fonctionnelle et programmation impérative (suite)

## Différences et avantages

- **Clarté :**

La version fonctionnelle utilise **filter** et **lambda**, ce qui exprime clairement l'intention de filtrer les nombres pairs.

- **Moins de code :**

La **version fonctionnelle** est **plus concise**, ce qui **réduit la quantité de code** à lire et à comprendre.

- **Éviter les effets de bord :**

Dans la **version impérative**, il y a une **variable mutable** (`pairs`), tandis que **la version fonctionnelle ne modifie pas d'état externe**.

- **Réutilisabilité :**

Les fonctions comme **filter** peuvent être **facilement réutilisées** dans d'autres contextes, ce qui **favorise la modularité**.

➡ La programmation fonctionnelle peut **simplifier la logique de traitement des données**, rendant le **code plus lisible** et **moins sujet aux erreurs en évitant les effets de bord**.

Les **effets de bord** (ou "**side effects**" en anglais) se réfèrent aux **modifications de l'état d'un programme** qui se produisent **en dehors d'une fonction ou d'un bloc de code**.

En d'autres termes, lorsque l'exécution d'une fonction modifie quelque chose d'autre que ses entrées, on parle d'effet de bord.

## Exemples d'effets de bord :

### 1. Modification de variables globales

```
compteur = 0

def incrementer():
    global compteur
    compteur += 1  # Effet de bord : modification d'une variable globale

incrementer()
print(compteur)  #Affiche 1
```

## Exemples d'effets de bord (suite) :

### 2. Modification de structures de données

```
def ajout(liste, element):  
    liste.append(element)    # Effet de bord : modification de la liste  
                             passée en paramètre  
  
ma_liste = [1, 2, 3]  
ajout(ma_liste, 4)  
print(ma_liste)
```



## Chapitre 4 : La programmation fonctionnelle

- ☐ Introduction à la programmation fonctionnelle
- ☒ **Fonction pure**
- ☐ Fonction de première classe
- ☐ Fonction imbriquée et closures
- ☐ Fonction d'ordre supérieur
- ☐ Fonction anonyme `lambda`
- ☐ Fonctions natives `map, filter, zip`
- ☐ Fonction récursive

**Fonction pure**

# Fonction Pure

- Une **fonction pure** est une fonction qui ne modifie pas l'état global ni ne produit d'effets secondaires.
  - Elle renvoie toujours le même résultat pour les mêmes entrées.
  - Elle est déterministe et n'a pas de dépendance aux variables externes.
- Cœur de la programmation fonctionnelle, car elle garantit l'absence d'effets de bord.
- Facilite les tests et permet d'écrire du code immuable.

Exemple :

## Fonction impure



```
# Variable globale
compteur = 0

def impure_addition(a, b):
    global compteur
    compteur += 1
    return a + b
```

# Effet de bord : modification de la variable globale



## Fonction pure

```
def addition(a, b):
    return a + b
```

# Fonction de première classe

# Fonction de première classe

- ❑ Les fonctions de première classe peuvent être traitées comme une valeur de première classe, c'est-à-dire qu'elle peut être **affectées à des variables**,
  - ❑ **passées en tant qu'arguments** à d'autres fonctions, et
  - ❑ **retournées en tant que résultats** à partir d'autres fonctions.
- En Python, les fonctions sont des objets de première classe.
- La programmation fonctionnelle traite les fonctions comme des valeurs. Ce qui permet une écriture plus concise, modulaire et réutilisable du code.

## Exemple 1 : **affectées à des variables**

```
# Définition d'une fonction
def multiplier(x):
    return x * 2

# Affectation à une variable
fct_variable = multiplier

# Utilisation de la fonction stockée dans la variable
resultat = fct_variable(5)  # 10
```

multiplier est une  
fonction de première  
classe

# Fonction de première classe (suite)

## Exemple 2 :

```
def appliquer(fonction, valeur):
```

```
    return fonction(valeur)
```

retournées en tant que résultats à partir d'autres fonctions  
→ Retourne une valeur

```
result2 = appliquer(multiplier, 3)  
print(result2) #6
```

passées en tant qu'arguments à d'autres fonctions

```
result3 = appliquer(fct_variable, 4)  
print(result3) #8
```

multiplier est une  
fonction de première  
classe

# Fonction imbriquée (interne) et closures

# Fonction imbriquée (interne) et closures

- Une **fonction imbriquée** est une fonction **définie à l'intérieur d'une autre fonction**.
- La fonction imbriquée **peut accéder aux variables locales** de la fonction englobante, créant ainsi une portée lexicalement imbriquée.
- Elle est utile pour encapsuler du code ou pour **créer des fermetures (closures)**.

Exemple :

La fonction `interne` peut accéder aux variables de la fonction `externe` !

```
def externe(x):  
    def interne(y):  
        return x + y  
    return interne
```

`interne` est une  
fonction imbriquée  
(interne)

```
f = externe(5)  
res = f(3)  
print(res) #8
```

- **Closure=** la fonction `interne` fait référence aux variables de la fonction `externe`!
- `f` est une closure ; c'est la fonction qui a capturé `x=5` dans cet exemple même après que `externe` a terminé son exécution.
- Cette valeur de `x` est maintenue lorsque `interne` est appelée via `f`.



# Closures (Fermetures) :

- Une **fermeture (closure)** est une fonction imbriquée qui capture des variables locales de la fonction englobante même après que la fonction englobante ait terminé son exécution.
- Elle conserve un "clos" sur les variables de la portée de la fonction englobante, ce qui permet d'accéder à ces variables même après que la fonction englobante ait été appelée.

## Exemple :

```
def creer_fonction_multiplier(n):  
    def multiplier(x):  
        return x * n  
    return multiplier
```

# À ce stade, doubler est une fonction qui conserve la valeur 2 (de n) en mémoire.

```
doubler = creer_fonction_multiplier(2)  
resultat = doubler(5) # résultat est maintenant égal à 10
```

- Dans cet exemple, **creer\_fonction\_multiplier** prend un argument *n* et retourne une fonction **multiplier** qui prend un autre argument *x*.
- Lorsque nous appelons **creer\_fonction\_multiplier(2)**, elle crée une **closure** pour multiplier avec *n* égal à 2.
- Par conséquent, lorsque nous appelons **doubler(5)**, il utilise toujours la valeur 2 pour *n*, ce qui donne un résultat de 10.

**Fonction d'ordre supérieur**

# Fonction d'ordre supérieur

- Une **fonction d'ordre supérieur** est une fonction qui **prend une ou plusieurs fonctions comme arguments** et/ou **renvoie une fonction comme résultat**.
- Elle permet de **passer des fonctions en tant qu'arguments** ou de **retourner des fonctions à partir d'autres fonctions**.

Exemple :

```
def increment(x):  
    return x+1
```

```
def appliquer(f):  
    def calculer(x):  
        result= f(x)  
        return result  
    return calculer
```

```
res=appliquer(increment)  
print(type(res)) #<class 'function'>  
print(res(5))    #6
```

appliquer est une fonction  
d'ordre supérieur

Passage d'une ou plusieurs fonctions en tant  
qu'arguments

Retour de fonction(s) en tant que résultat(s)

**Fonction anonyme** `lambda`

# Fonction anonymes **lambda**

- La fonction **lambda** en Python est utilisée pour créer des fonctions anonymes (*fonction déclarée sans nom*) d'une seule ligne. Appelée aussi **fonction en ligne**.

Syntaxe d'une fonction lambda en python :

**lambda** arguments: expression

- La fonction **lambda** est utilisée lorsque la fonction est définie par une expression simple :

## Exemple 1 :

```
def f(x):  
    return x*2
```



```
f = lambda x:x*2  
print(f) # <function <lambda> at 0x0000028811262310>  
print(f(3)) #6
```

```
(lambda x: x*2)(3) #6
```

## Exemple 2 :

```
f = lambda x: 'Voyelle' if x.lower() in 'aeiou' else 'Consonne'  
print(f) #<function <lambda> at 0x000001CCB380DF80>  
print(f('a')) #Voyelle
```

## Exemple 3 :

```
result = (lambda x,y: len([i for i in range(x, y) if i % 2 == 0]))(2, 10)  
print(result) #4 #Affiche le nombre de nombres pairs entre 2 et 9
```

Écrire une fonction anonyme qui permet de supprimer à partir d'un mot donné tous les chiffres de 0 à 9. Tester cette fonction

**Exemple d'exécution :**

```
f("Hello123") retourne 'Hello'
```

Fonctions natives **map**, filter, zip

# Fonctions natives `map`, `filter`, `zip`

La fonction `map()` : utilisé pour **transformer** chaque élément d'un itérable en appliquant une fonction.

## Syntaxe de la fonction `map`

```
map(fct, it)
```

→ Appliquer la fonction `fct` à chaque élément de l'itérable `it` (comme une liste ou un tuple) et retourner un nouvel itérable avec les résultats de cette fonction.

## Exemple 1 :

Objectif : Obtenir une nouvelle liste en doublant chaque élément d'une liste donnée.

Utiliser une Liste en compréhension  
comme retour d'une fonction

```
def doubleStaff(liste):  
    return [v*2 for v in liste]  
  
nombre = [1, 2, 3, 4]  
print(doubleStaff(nombre))  
#[2, 4, 6, 8]
```

Utiliser la fonction `map()` pour doubler chaque nombre dans  
une liste.

```
nombres = [1, 2, 3, 4]  
result = map(lambda x: x * 2, nombres)  
print(result) #<map object at 0x000001CCB38145B0>  
print(type(result)) #<class 'map'>  
print(list(result)) #[2, 4, 6, 8]
```




# Fonctions natives `map`, `filter`, `zip` (suite)

## Exemple 2 :

*Objectif :* Utiliser la fonction **`map()`** pour additionner les éléments correspondants de deux listes même si les listes ont des tailles différentes.

```
liste1 = [1, 2, 3, 4, 5]
liste2 = [1, 2, 1]
```



```
[2, 4, 4]
```

```
liste1 = [1, 2, 3, 4, 5]
liste2 = [1, 2, 1]
liste = map(lambda x,y:x+y, liste1, liste2)
print(list(liste)) #[2, 4, 6, 8]
```

Ou bien

```
def addition(x, y):
    return x + y

liste1 = [1, 2, 3, 4, 5]
liste2 = [1, 2, 1]
liste = map(addition, *[liste1, liste2])
print(list(liste)) #[2, 4, 6, 8]
```

**Fonctions natives** `map`, **`filter`**, `zip`

# Fonctions natives `map`, `filter`, `zip`

- La fonction `filter()` permet de **filtrer un itérable** (comme une liste, un tuple, etc.) **en fonction d'une fonction de filtrage** donnée.
- Elle retourne un nouvel itérable contenant seulement les éléments pour lesquels la fonction renvoie `True`.

## Syntaxe de la fonction `filter`

```
filter(function, iterable)
```

### Exemple 1 :

Objectif : Utiliser la fonction `filter()` pour garder uniquement les nombres pairs dans une liste.

```
nombres = [1, 2, 3, 4, 5, 6]
pairs = filter(lambda x: x % 2 == 0, nombres)
print(pairs) #<filter object at 0x000001CCB3852FE0>
print(type(pairs)) #<class 'filter'>
print(list(pairs)) #[2, 4, 6]
```

# Fonctions natives `map`, `filter`, `zip`

## Exemple 2 :

Objectif : Utiliser la fonction `filter()` pour garder uniquement les nombres pairs dans une liste.

```
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False

# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function
filtered = list(filter(fun, sequence))

print("The filtered letters are: ", filtered)
```

Modifier ce code :  
Utiliser la fonction  
`lambda` au lieu de la  
fonction `fun`

Après exécution : `>>> The filtered letters are: ['e', 'e']`

**Fonctions natives** `map, filter, zip`

# Fonctions natives `map`, `filter`, `zip`

- La fonction **`zip()`** est utilisée pour **regrouper** des **éléments** de **plusieurs itérables** en un **seul itérable de tuples**.
- Elle **renvoie** un **objet itérable** de tuples (`<class 'zip'>`)
- La longueur de l'itérable de sortie sera égale à la longueur de la séquence la plus courte parmi les séquences d'entrée.

## Syntaxe de la fonction `zip`

```
zip(*iterables)
```

### Exemple 1 :

```
noms = ["Khaled", "Sami", "Mouna"]
ages = [25, 30, 35]

# Utilisation de zip pour associer les éléments de noms et ages
resultat = zip(noms, ages)
print(type(resultat)) #<class 'zip'>

# Affichage du résultat sous forme de liste
print(list(resultat)) #[('Khaled', 25), ('Sami', 30), ('Mouna', 35)]
```

# Fonctions natives `map`, `filter`, `zip` (suite)

Essayez de voir aussi cet exemple 😊

## Exemple 2 :

```
liste1 = [1,2,3]
liste2 = [5,6,7]
liste3 = ['a','b']
ch="def"

liste = list(zip(liste1, liste2, liste3, ch))
print(liste) #[(1, 5, 'a', 'd'), (2, 6, 'b', 'e')]
```

# Fonction récursive



# Fonction récursive

Une fonction s'appelle elle-même de manière itérative pour résoudre un problème!

## Syntaxe de la fonction zip

```
def fonction_recursive( parametres ):  
    if..... : # condition d'arrêt : clause ou cas de base  
        return..... # terminaison de l'algorithme  
    else:  
        return..... # instructions contenant un appel à fonction_recursive
```

```
def somme(n):  
    if n <= 0:  
        return 0  
    else:  
        return n + somme(n-1)  
  
result = somme(10)  
print(result)
```

Pile		
somme(0)=0		
1 + somme(0)	1+0=1	
2 + somme(1)	2+1=3	
3 + somme(2)	3+3=6	
4 + somme(3)	4+6=10	
5 + somme(4)	5+10=15	
6 + somme(5)	6+15=21	
7 + somme(6)	7+21=28	
8 + somme(7)	8+28=36	
9 + somme(8)	9+36=45	
10 + somme(9)	10+45=55	

- **Fonction pure** = fonction sans effets de bord = ne modifie pas l'état de variables = produit toujours la même sortie pour les mêmes entrées
- **Fonction anonyme (lambda)** = fonctions n'ont pas de nom explicite
- **Fonction de première classe** = fonction traitée comme une valeur (**affectée à des variables**) ou **passée en tant qu'argument** à d'autres fonctions ou **retournée en tant que résultat**
- **Fonction d'ordre supérieur** = fonction peut prendre ou retourner d'autres fonctions
- **Fonction imbriquée (interne)** = fonction définie à l'intérieur d'une autre fonction
- **Closure** : Une closure est une fonction qui capture des variables de son environnement extérieur (variables définies dans la fonction parent) et qui utilise des variables de cet environnement plus tard, même si la fonction parente n'est plus en cours d'exécution.

**Module `itertools` Travailler avec des itérables!**

# Module `itertools`

- Le module `itertools` propose des **fonctions** prêtes à l'emploi qui **retournent des itérateurs "paresseux"** (qui génèrent des éléments à la demande), **comme les générateurs**. Ces fonctions sont **faciles à utiliser** et **bien optimisées**.
  - Les générateurs en Python sont des fonctions qui génèrent des éléments à la volée avec `yield` et peuvent être utilisés pour économiser de la mémoire.
- ➡ Les deux (`itertools` et générateurs) sont très pratiques quand on veut travailler **avec de grandes quantités de données sans épuiser la mémoire**.

# Module `itertools` (suite)

Fonction	Description
<code>starmap(f, it)</code>	Applique la fonction <code>f</code> à chaque élément de l'itérable <code>it</code> , en décompressant les arguments. Les éléments de <code>it</code> doivent être des tuples ou des listes.
<code>filterfalse(f, it)</code>	Filtre les éléments de l'itérable <code>it</code> et garde ceux pour lesquels la fonction <code>f</code> retourne <code>False</code> . ( <i>l'opposé de <code>filter()</code></i> )
<code>takewhile(f, it)</code>	Prend les éléments de <code>it</code> tant que la fonction <code>f</code> retourne <code>True</code> . Dès que <code>f</code> retourne <code>False</code> , il arrête la collecte.
<code>dropwhile(f, it)</code>	Ignore les éléments de <code>it</code> tant que <code>f</code> retourne <code>True</code> . Dès que <code>f</code> retourne <code>False</code> , elle commence à retourner les éléments restants.
<code>groupby(it, key=None)</code>	Grouper les éléments consécutifs de <code>it</code> qui ont la même clé <code>key</code> (par défaut, les éléments identiques).

Exemple : `from itertools import *`

```
pairs = [(1, 2), (3, 4), (5, 6)]
print(list(starmap(lambda x,y:x+y, pairs)))#[3, 7, 11]
```

**#Attention!!!**

```
pairs = [[1, 2], [3, 4], [5]]
print(list(starmap(lambda x,y:x+y, pairs)))
```

```
nombres = [1, 2, 3, 4, 5, 6]
print(list(filterfalse(lambda x:x%2==0, nombres))) #[1, 3, 5]
```

```
nombres = [1, 2, 3, 4, 5, 6, 7]
print(list(takewhile(lambda x:x<5, nombres)))
#[1, 2, 3, 4]
```

```
nombres = [1, 2, 3, 4, 5, 6, 7]
print(list(dropwhile(lambda x:x<5, nombres)))
#[5, 6, 7]
```

```
nombres = [1, 2, 2, 3, 3, 3, 4, 4]
result = groupby(nombres)
for clé, groupe in result:
    print(f'Clé: {clé}, Groupe: {list(groupe)}')
```

```
Clé: 1, Groupe: [1]
Clé: 2, Groupe: [2, 2]
Clé: 3, Groupe: [3, 3, 3]
Clé: 4, Groupe: [4, 4]
```

```

from itertools import *

pairs = [(1, 2), (3, 4), (5, 6)]
print(list(starmap(lambda x,y:x+y, pairs)))#[3, 7, 11]

print(list(map(lambda el: el[0] + el[1], pairs))) #[3, 7, 11]

#Attention!!!
pairs = [[1, 2], [3, 4], [5]]
print(list(starmap(lambda x,y:x+y, pairs)))
print(list(starmap(lambda x,y=2:x+y, pairs)))# [3, 7, 11]
#TypeError: <lambda>() missing 1 required positional argument: 'y'

nombres = [1, 2, 3, 4, 5, 6]
print(list(filterfalse(lambda x:x%2==0, nombres))) #[1, 3, 5]
print(list(filter(lambda x:x%2!=0, nombres)))
print([x for x in nombres if x % 2 != 0])

nombres = [1, 2, 3, 4, 5, 6, 0, 7]
print(list(takewhile(lambda x:x<5, nombres))) #[1, 2, 3, 4]

nombres = [1, 2, 3, 4, 5, 6, 0, 7]
print(list(dropwhile(lambda x:x<5, nombres))) #[5, 6, 0, 7]

nombres = [1, 2, 2, 3, 3, 3, 4, 4]
result = groupby(nombres)
for clé, groupe in result:
    print(f'Clé: {clé}, Groupe: {list(groupe)}')

```

- `groupby` permet de regrouper les éléments d'un itérable en fonction d'une clé fournie par une fonction.
- en Python, `groupby` nécessite que les **éléments de l'itérable soient triés avant de les grouper**.

Vous disposez d'une *liste de tuples*, où chaque tuple représente un étudiant avec son *prénom et son âge*. Par exemple :

```
data = [('Ahmed', 25), ('Mohamed', 30), ('Nadjib', 35), ('Mouna', 30), ('Sami', 25)]
```

## Travail à faire :

1. Trier la liste des étudiants par âge.
2. Grouper les étudiants par âge à l'aide de `itertools.groupby`.

Résultat  
d'exécution :

```
25 : [('Ahmed', 25), ('Sami', 25)]
30 : [('Mohamed', 30), ('Mouna', 30)]
35 : [('Nadjib', 35)]
```

# Principes clés de la programmation fonctionnelle

RECAP.

La programmation fonctionnelle repose sur :

- 1.Immutabilité** : Les données ne changent pas une fois créées.
- 2.Absence d'effets de bord** : Une fonction agit uniquement sur ses entrées et retourne une sortie.
- 3.Composabilité** : Les fonctions sont des blocs modulaires que l'on combine pour résoudre des problèmes.
- 4.Traitement des fonctions comme des valeurs de première classe** : Les fonctions peuvent être assignées à des variables, passées en argument ou retournées par d'autres fonctions.

les types de fonctions utilisés en programmation fonctionnelle incluent **les fonctions pures**, **les fonctions de première classe**, **les fonctions d'ordre supérieur**, **les fonctions imbriquées** (ou closures), les fonctions **lambda**, les **fonctions récursives**, ainsi que les **fonctions natives** comme `map`, `filter`, et `zip`.

Le module `itertools` est un excellent complément à la programmation fonctionnelle en Python, car il fournit des **outils pour créer des itérateurs efficaces et paresseux** (Lazy Iterators), souvent utilisés avec des fonctions comme `map`, `filter`.



# Activité 1

Étant donné le tuple `participants` qui représente la base des participants :

```
participants=(['p001', ['Amir BA', (1,2,4), 101]],  
              ['p002', ['Ali BS', (1,2), 101]],  
              ['p003', ['Salma B', (1,), 104]],  
              ['p004', ['Said G', (2,), 103]],  
              ['p005', ['Salma D', (1,2,3), 106]],  
              ['p006', ['Fatma B', (1,3), 105]],  
              ['p007', ['Kahled M', (2,3), 101]],  
              ['p008', ['Heni A', (2,3), 102]],  
              ['p009', ['Edam K', (1,2,3), 106]],  
              ['p010', ['Sahar FA', (3,), 105]],  
              ['p011', ['Salima D', (3,), 104]])
```

Chaque participant est caractérisé par un `code`, son *Nom* et *prénom* (`name`), les *numéros de sessions* (`n_sess`) spécifiques ou les ateliers auxquels le participant s'est inscrit et son *numéro de chambre* (`n_ch`)

**NB.** Le traitement de toutes les fonctions doit être effectué sans faire recours à une structure itérative

# Activité 1 (suite)

1/ Écrire une **fonction génératrice** nommée **gen\_participant(p=participants)** qui prend en paramètres le tuple de participants **p** (par défaut **p=participants**) et renvoie un dictionnaire pour chaque participant. Le format de chaque élément du dictionnaire devrait être le suivant :

```
{ 'code': [name, (n_sess1, n_sess2, ...), n_ch] }
```

```
def gen_participant(p=participants):
```

```
.....
```

```
g=gen_participant()
print(next(g))#{'p001': ['Amir BA', (1, 2, 4), 101]}
print(next(g))#{'p002': ['Ali BS', (1, 2), 101]}
```

```
participants=(['p001',['Amir BA', (1,2,4),101]],
               ['p002',['Ali BS', (1,2),101]],
               ['p003',['Salma B', (1,),104]],
               ['p004',['Said G', (2,),103]],
               ['p005',['Salma D', (1,2,3),106]],
               ['p006',['Fatma B', (1,3),105]],
               ['p007',['Kahled M', (2,3),101]],
               ['p008',['Heni A', (2,3),102]],
               ['p009',['Edam K', (1,2,3),106]],
               ['p010',['Sahar FA', (3,),105]],
               ['p011',['Salima D', (3,),104]])
```

# Activité 1 (suite)

2/ Définir une fonction **grouper\_chambre(g=gen\_participant)** qui accepte une fonction **g** des participants (par défaut **g=gen\_participant**). La fonction doit renvoyer un dictionnaire où chaque clé correspond à un numéro de chambre, et chaque valeur est une chaîne de caractères regroupant le(s) nom(s) et prénom(s) des résidents assignés à cette chambre, séparés par **" / "**

```
g=gen_participant()
print(next(g))#{'p001': ['Amir BA', (1, 2, 4), 101]}
print(next(g))#{'p002': ['Ali BS', (1, 2), 101]}
```

```
def grouper_chambre(g=gen_participant):
```

```
.....
```

```
grouper_chambre(g=gen_participant)
{101: 'Amir BA/Ali BS/Kahled M',
 102: 'Henri A',
 103: 'Said G',
 104: 'Salma B/Salima D',
 105: 'Fatma B/Sahar FA',
 106: 'Salma D/Edam K'}
```



# Fin Chapitre 4

**Fatma Ben Said**

[fatma.bensaid@iit.ens.tn](mailto:fatma.bensaid@iit.ens.tn)