

# Modularité et initiation à la Programmation Orientée Objet

Quand nous utilisons certaines fonctions où certains objets Python, qu'ils soient *built-in* ou bien importés à partir de *modules*, nous nous posons rarement la question de savoir quelle est leur **implémentation**, c'est-à-dire la manière dont-ils ont été conçu et programmé. Nous faisons *globalement confiance* aux concepteurs du langage ou du module.

Ce qui nous importe est plutôt **l'interface** de ces objets, c'est-à-dire la façon dont nous pouvons interagir avec ces objets : les créer, les affecter, les additionner, les supprimer,...

Dans cette partie nous verrons comment créer un module, le documenter, et définir une interface claire. Nous verrons les prémices d'un nouveau **paradigme de programmation** : la Programmation Orientée Objet(**POO**).

La suite de cette partie est grandement inspirée de [Numériques et Sciences Informatique, 24 leçons avec exercices corrigé, Ellipse](#)

## 1. Un premier problème



### Abstract

Voici une propriété probabiliste peu intuitive : il suffit d'avoir un groupe de 23 personnes pour que la probabilité que deux personnes aient la même date d'anniversaire soit supérieure à 50%.

Nous allons construire un programme Python qui permettra de vérifier expérimentalement cette propriété.

Pour modéliser le problème :

- plutôt que d'utiliser des dates, nous allons utiliser des entiers de 1 à 365 ;
- nous allons créer une fonction *sans paramètres* `genere_groupe()` qui renvoie un tableau aléatoire de 23 entiers de 1 à 365 ;
- nous allons créer une fonction `contient_doublon(t)` qui renverra `True` si le tableau contient un doublon, et `False` sinon ;
- puis nous créerons une fonction `teste_hypothese(n)` qui testera sur un échantillon de `n` groupes la présence d'un doublon ou non, et renverra le nombre de groupes ayant eu des doublons.

## ? Exercice

Créer un fichier `recherchesDates.py` et **implémenter** les fonctions précédentes. Des solutions sont proposées dans les parties ci-dessous, mais vous **devez d'abord tester par vous-mêmes**. Vous pouvez cependant utiliser les indices ci-dessous pour vous aider

### Procédure `genere_groupe()`

1. Utiliser la fonction `randint` du module `random` (voir la [doc](#))
2. Utiliser les méthodes de listes (voir la [doc](#))

### Fonction `contient_doublon(t)`

Une possibilité est de créer d'abord une liste vide `vus`, dans laquelle on ajoutera les valeurs déjà vue lors du parcours de la liste `t`.

Ainsi, on parcourt la liste `t`

1. si l'élément est dans `vus`, c'est qu'il y a un doublon, donc on arrête la fonction en renvoyant `True`
2. si l'élément n'est pas dans `vus`, c'est donc la première fois qu'on le voit, et on l'ajoute à `vus`.
3. Si on atteint la fin de la liste, c'est qu'il n'y a pas de doublons.

### fonction `teste_hypothese(n)`

Il faut :

1. Initialiser une variable comptant le nombre de doublons à 0.
2. Effectuer `n` fois une boucle qui :
  - a. Génère un groupe aléatoire.
  - b. Incrémmente de 1 le compteur si ce groupe contient un doublon
3. Renvoie le compteur.

## i Solution

```

1 from random import randint
2
3 def genere_groupe() -> list :
4     """fonction renvoyant un tableau de 23 nomb
5     return [randint(1,365) for _ in range(23)]
6
7 def contient_doublon(t : list) -> bool :
8     """fonction renvoyant un booléen signalant
9     s = [] # s est un tableau temporaire conten
10    for data in t :
11        if data in s : # si data est déjà dans
12            return True
13        else : # sinon on ajoute data à la lis
14            s.append(data)
15    return False
16
17 def teste_hypothese(n : int) -> int:
18     """fonction renvoyant le nombre de groupes
19     sur un échantillon de n groupes"""
20     nbDoublons = 0
21     for _ in range(n) :
22         t = genere_groupe()
23         if contient_doublon(t) :
24             nbDoublons +=1
25     return nbDoublons
26

```





## Preuve mathématique



Cette preuve est donnée à titre indicatif, et n'a ni à être connue, ni même à être comprise.

Considérons notre groupe de 23 personnes, et cherchons la probabilité que les 23 personnes **n'aient pas la même date anniversaire** :

- la première peut avoir n'importe quel date anniversaire, donc 365 possibilité sur 365 dates possibles.
- La deuxième ne peut pas avoir la même date que les deux premiers, donc 364 possibilités sur 365.
- La troisième ne peut avoir la même date que les deux premiers, donc 363 possibilités sur 365.
- ...
- La                    ne peut avoir la même date que les                    précédents, donc                    possibilités.
- ...
- La 23ème ne peut avoir la même date que les 22 précédents, donc                    possibilités.

La probabilité cherchée est donc                    —                    —                    —                    —                    où                    est la factorielle de 365, soit la multiplication

Or l'événement contraire de "les 23 personnes n'ont pas la même date anniversaire" est l'événement "au moins 2 personnes parmi les 23 ont la même date d'anniversaire". Donc sa probabilité est                    soit en calculant environ                    , soit                    \%.

Plus d'informations peuvent être trouvées sur l'[article correspondant de wikipedia](#).

## 2. Différentes solutions ?

Bien entendu, les solutions proposées ci-dessus ne sont pas uniques. Elles sont mêmes **non optimales** (en tout cas pour la fonction `contient_doublon(t)`). Il est tout à fait possible de proposer d'autres **implémentations** du code, c'est-à-dire **d'autres façons de coder** la fonctionnalité voulue. Ainsi on pourrait regarder les implémentations suivantes, et les comparer entre elles :

## ? Exercice : autres implémentations de `contient_doublon(t)`

### Tableau de booléens

```
def contient_doublon(t) :
    """fonction renvoyant un booléen signalant la présence ou non d'un doublon dans le tableau"""
    s = [False]*365 # s est un tableau temporaire contenant false pour chaque date
    for data in t :
        if s[data] : # si s[data] est vrai (True), alors il y a doublon
            return True
        else : # sinon on bascule s[data] à True
            s[data] = True
    return False
```

C'est une solution simple. Mais que dire de ses avantages et de ses inconvénients ?

### Tableau de bits

```
def contient_doublon(t) :
    """fonction renvoyant un booléen signalant la présence ou non d'un doublon dans le tableau"""
    s = 0
    for data in t :
        if s & (1<<data) != 0 :
            return True
        else :
            s = s | (1<<data)
    return False
```

C'est une solution beaucoup plus complexe (et hors programme de Terminale dans sa conception). Quels sont ses avantages et ses inconvénients ?

### Table de hachage

```
def contient_doublon(t) :
    """fonction renvoyant un booléen signalant la présence ou non d'un doublon dans le tableau"""
    s = [[] for _ in range(23)]
    for data in t :
        if data in s[data%23] :
            return True
        else :
            s[data%23].append(data)
    return False
```

## ✓ Solution



### Solution originale



L'avantage est la simplicité du code, et c'est à peu près tout... Par contre les inconvénients sont nombreux, en particulier le **coût en temps** : en effet à chaque tour de boucle `for data in t`, on exécute l'instruction `data in s`, qui parcourt tout le tableau `s`... On a donc une complexité en temps en  $O(n^2)$  (au pire). Pour un tableau de 23 éléments, c'est acceptable, mais dans l'hypothèse d'un tableau de plus grande taille, c'est absolument à éviter !

### Solution tableau de booléens

Un des avantages est que la complexité en temps est bien meilleure que pour la première solution. Il n'y a plus les deux boucles imbriquées, d'où un gain considérable. Cependant on peut avoir un problème de **coût en mémoire**, car on utilise un tableau de taille 365 pour uniquement vérifier 23 dates. Dans le cadre d'une comparaison sur un ensemble de valeurs possibles supérieures à 365, le coût en mémoire peut vite devenir problématique.

### Solution tableau de bits

La solution est très complexe, mais elle a un grand mérite : un booléen, en python, est en fait un **entier** (0 ou 1), donc stocké sur... **8 octets** ! (source [ici](#)) Or il n'est pas nécessaire d'utiliser 8 octets, soit 64 bits, pour stocker un booléen... En fait il suffit d'un seul bit ! Cette solution divise donc par **64** la taille mémoire par rapport à la solution précédente !

C'est globalement un bon avantage dans cette situation,; mais cela reste rapidement insuffisant si le nombre d'éléments auquel on s'intéresse est bien plus grand que 365.

Il faut noter que le **tableau de bits** (ou *bit set* ou *bit array*) est une structure compacte qui permet de représenter facilement des tableaux de booléens. Elle permet une meilleure utilisation des ressources mémoires dans les cas où celle-ci est limitée, comme par exemple dans la mémoire cache du processeur.

### Solution table de hachage

Comme nous l'avons vu en classe de première, la table de hachage est une solution efficace et élégante qui permet de gagner à la fois du **coût en temps** (on ne parcourt pas un tableau, on atteint directement l'objet par sa *clé*, ou en tout on parcourt un sous-ensemble beaucoup plus petit), et du **coût en mémoire** (le tableau des clés est de la taille strictement nécessaire).

## 3. Une même interface

## ? Exercice

Quand on observe les 4 propositions de codes pour la fonction `contient_doublon(t)`, on peut constater que ces 4 codes sont quasiment identiques. Quelles sont ces parties identiques ?

## ✓ Solution



```
def contient_doublon(t) :
    """fonction renvoyant un booléen signalant la présence ou non d'un doublon dans le tableau"""
    s = ...
    for data in t :
        if ... :
            return True
        else :
            ...
    return False
```

Les parties en pointillé de la solution précédente vérifient les conditions suivantes :

- `s` représente un ensemble de date, et le premier trou correspond à la création de cette structure.

- Le deuxième trou consiste à vérifier si `data` est contenu dans `s`.
- le troisième trou consiste à ajouter `data` à `s`

Seules ces trois parties changent dans les 4 programmes.

On pourrait alors isoler ces trois aspects dans trois fonctions différentes et obtenir le code *factorisé* suivant :

#### ✓ Code factorisé

```
def contient_doublon(t) :  
    """fonction renvoyant un booléen signalant la présence ou non d'un doublon dans le tableau"""  
    s = cree()  
    for data in t :  
        if contient(data,s) :  
            return True  
        else :  
            ajoute(data,s)  
    return False
```

On définit ainsi une fonction `contient_doublon(t)` **complètement séparée** de la représentation de la structure `s`.

Le ou la programmeur·euse qui souhaite simplement utiliser la structure de donnée `s` **n'a pas à se préoccuper** de la façon dont elle a été **implémentée**. Il ou elle n'a besoin que de connaître son **interface** :

- la fonction `cree()` sert à construire une structure ;
- la fonction `contient(data,s)` sert à regarder si `data` est contenu dans la structure `s` ;
- La fonction `ajoute(data,s)` ajoute l'élément `data` à la structure `s`.

C'est exactement ce qui se passe quand on utilise des modules python : on ne cherche pas à savoir *comment sont programmés* les fonctions du modules(c'est-à-dire **l'implémentation du module**) - car on fait confiance aux programmeur·euse·s de ce module, mais juste à savoir *comment utiliser* ces fonctions(= **l'interface du module**).

Encore mieux, le ou la programmeur·euse du module peut, si il ou elle ne change pas l'**interface** (c'est-à-dire la manière *d'utiliser* les fonctions), améliorer ces fonctions (en temps, en mémoire, etc...) sans même que l'utilisateur·trice n'ait à changer quoi que ce soit à son propre programme, qui continuera à fonctionner (mieux, du moins on espère...).