Chap. 2 — Modularité

1.1 — Un exemple : le paradoxe des anniversaires

[14]:
```python
# Programme 1
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = set()
    for x in t:
        if x in s:
            return True
        s.add(x)
    return False
```

[15]:
```python
# Programme 2
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = []
    for x in t:
        if x in s:
            return True
        s.append(x)
    return False
```

[16]:
```python
# Programme 3
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = [False] * 366
    for x in t:
        if s[x]:
            return True
        s[x] = True
    return False
```

1.2 — Factorisation du code

Les trois programmes précédents se ressemblent beaucoup :

- ils font la même chose, mais avec des stratégies complètement différentes
- ils ont la même structure.

Cette structure peut d'ailleurs se résumer en :

-`s` représente d'une manière ou d'une autre un ensemble de dates qu'il faut *créer*. - Il faut vérifier si `s` *contient* l'élément `x`. - Il faut être capable d'*ajouter* l'élément `x` à `s` si besoin.

Ce qui donne, en délégant ces trois aspects aux fonctions `cree()`, `contient()` et `ajoute()` :

```
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = cree()
    for x in t:
        if contient(s,x):
            return True
        ajoute(s,x)
    return False
```

Ainsi :

- pour changer le mode de représentation des dates, il ne faut plus changer `contient_doublon()`
- l'ensemble de dates peut être réutilisés dans d'autres programmes
- il y a séparation entre le programme qui utilise les dates et les programmes qui définissent comment sont programmées en interne ces dates.

Ces trois fonctions représentent l'**interface** entre le programme qui utilise l'ensemble de dates **et** les programmes qui définissent d'une façon ou d'une autre cet ensemble.

1.3 Modules

Une des clés du développement à grande échelle consiste à séparer proprement les différentes parties d'un programme.

Par exemple on peut séparer la définition d'une structure de données (comme l'ensemble de dates) et son utilisation.

On peut aussi séparer la partie interface graphique d'une application de sa partie logique qui en constitue le cœur.

fonction	description
`cree()`	crée et renvoie un ensemble de dates vide
`contient(s,x)`	renvoie `True` si et seulement si l'ensemble `s` contient la date `x`
`ajoute(s,x)`	ajoute la date `x` à l'ensemble `s`

Chaque morceau de code peut être placé dans un fichier différent appelé **module**.

Pour importer les fonctions définies dans un module et les utiliser, il faut utiliser le mot clé `import`

Par exemple, pour importer le module permettant de gérer les valeurs aléatoires, on importe `random` grâce à l'instruction `import random`. Auquel cas, pour utiliser la fonction `randint()` du module, il faut écrire `random.randint()`. Mais sous cette forme, cette façon d'importer est à éviter car ce sont *toutes* les fonctions du modules qui sont importées.

Il est préférable de n'importer que la ou les fonctions utiles. Par exemple `from random import randint` ne va importer que la fonction `randint(a,b)` qui permet de choisir aléatoirement un nombre entier compris entre `a` et `b` inclus.

Par exemple, pour créer son propre module, il suffit de sauvegarder dans un fichier `monModule.py` les fonctions.

Pour importer les fonctionnalités d'un module, il faut que le fichier `monModule.py` soit dans le même répertoire puis alors il suffit d'utiliser le mot clé `import` en écrivant : `from monModule import ...`.

Par exemple, les fonctions `cree()`, `ajoute()` et `contient()` peuvent être sauvegarder dans le fichier `dates.py`.

```
def cree():
    return [False] * 366

def contient(s,x):
    return s[x]

def ajoute(s,x):
    s[x] = True
```

Ensuite, si l'on souhaite utiliser notre ensemble de date dans un programme, il

suffit d'écrire `from date import cree, ajoute, contient` en ayant préalablement placé le fichier `date.py` dans le même répertoire que le fichier de travail `anniversaire.py` :

```
from dates import cree, contient, ajoute
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = cree()
    for x in t:
        if contient(s,x):
            return True
        ajoute(s,x)
    return False
```

Après avoir écrit votre module `date.py`, **écrire** un programme permettant de savoir combien d'élève il faut en moyenne dans une école pour qu'un anniversaire soit fêté chaque jour.

Pour cela, tirer au hasard des dates et les stocker dans un ensemble jusqu'à ce que toutes les dates aient été obtenues au moins une fois.

Répéter cette expérience 1000 fois et afficher une valeur moyenne.

[29]:
```
from dates import cree, contient, ajoute
from random import randint

def fete_continue():
    compteur = 0
    nombre_dates = 0
    s = cree()
    while nombre_dates < 365:
        compteur += 1
        x = randint(1,365)
        if not contient(s,x):
            nombre_dates += 1
            ajoute(s,x)
    return compteur

n = 0
for _ in range(1000):
    n += fete_continue()

print("En moyenne", n/1000, "élèves")
```

```
En moyenne 2388.026 élèves
```


1.4 Interfaces

Pour chaque module, on distingue :

- son **implémentation** : c'est-à-dire le code lui même et
- son **interface**, consistant en une énumération des fonctions définies dans le module qui sont destinées à être utilisées dans la réalisation d'autres modules, appelés *clients*.

L'interface doit expliciter ce qu'un utilisateur a besoin de connaître des fonctions proposées : *comment* et *pourquoi* les utiliser.

L'objectif est que :

1. ces fonctions soient suffisantes pour permettre à un utilisateur de faire appel aux fonctionnalités du module et
2. que ces fonctions soient utilisées sans avoir besoin d'aller consulter le code du module.

Pour chaque fonction il faut :

- un nom
- la liste des paramètres
- sa spécification, c'est-à-dire les conditions auxquelles la fonction peut être appliquée et les résultats à attendre.

La documentation de l'interface peut être vue comme un **contrat** entre l'auteur du module et ses utilisateurs.

C'est mieux si le nombre de choses à lire est limité, facile à comprendre et à mémoriser.

Par exemple, voici l'interface de l'ensemble de dates.