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() -> list` qui renvoie un tableau aléatoire de 23 entiers de 1 à 365 ;
- nous allons créer une fonction `contient_doublon(t : list) -> bool` qui renverra `True` si le tableau contient un doublon, et `False` sinon ;
- puis nous créerons une fonction `teste_hypothese(n : int) -> float` qui testera sur un échantillon de `n` groupes la présence d'un doublon ou non, et renverra le taux de groupes ayant eu des doublons sous la forme d'un pourcentage.

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() -> list` >

Fonction `contient_doublon(t:list) -> bool` >

fonction `teste_hypothese(n : int) -> float` >

Solution >

Preuve mathématique >

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`). 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 : list) -> bool :
    """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 : list) -> bool :
    """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 : list) -> bool :
    """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

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

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 : list) -> bool :
        """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/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 connaitre 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 module(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...).