Chapitre 7

programmation

Modularité, interface et encapsulation

« mots-dû-las-riz-thé ? »

À retenir

- on regroupe des fonctions communes à certaines fonctionnalités en *modules* ;
- la partie publique de chaque fonction, destinée à l'utilisateur, s'appelle l'*interface* et doit être convenablement documentée ;
- d'autres variables ou fonctions sont cachées à l'utilisateur, c'est le principe d'*encapsulation* ;
- une fonction ou un module peuvent avoir la même interface mais des *implémentations* (choix de programmation) différentes.

I Un problème, de multiples réponses

On considère un groupe de personnes, plus ce groupe est grand, plus il y a de chances que deux personnes soient nées le même jour de l'année. Le *paradoxe des* anniversaires énonce qu'à partir de 23 personnes, il y a plus de 50% de chances que deux personnes soient nées le même jour de l'année. Ce n'est pas un paradoxe au sens strict du terme, c'est simplement un résultat qui est contraire à l'intuition : la plupart des gens pensent qu'il faut bien plus que 23 personnes.
De la même manière, lorsqu'on considère des nombre entiers positifs codés sur 2 octets (donc compris entre 0 et $2^{16} - 1 = 65\,535$, mais cela revient au même de dire qu'ils sont compris entre 1 et 65 536), il suffit d'en prendre 302 au hasard pour qu'il y ait plus d'une chance sur 2 qu'il y ait au moins un doublon (une valeur revenant au moins une fois).
Par exemple dans ce petit échantillon de 302 entiers compris entre 1 et 2^{16} :

```
[13654, 26764, 60127, 56265, 45203, 54601, 23471, 64648, 49436, 32684, 4685, 61418, 8441, 10200,
29042, 55598, 35106, 59628, 16003, 52546, 61235, 61380, 58092, 15876, 41296, 5825, 11755, 46620, 33256,
```

```
21388, 34496, 50818, 24255, 21645, 59590, 46160, 29287, 28482, 7056, 62317, 7646, 48862, 580, 55506,
37346, 20788, 18739, 46029, 17621, 23795, 64827, 62778, 44784, 1732, 56030, 36325, 5513, 18255, 5423,
30071, 27916, 26456, 42655, 56515, 54266, 30311, 9712, 56000, 57606, 29080, 11732, 6675, 18147, 18031,
31923, 7587, 10177, 11595, 45194, 60765, 57430, 22114, 7692, 22005, 37297, 7817, 35883, 21041, 25233,
8245, 17171, 604, 15615, 49219, 5292, 61211, 32599, 27813, 59838, 15470, 35127, 19969, 15707, 60577,
28106, 54636, 18636, 10802, 45, 3962, 12676, 56137, 4457, 18793, 64445, 48181, 61137, 18182, 3579,
42258, 50192, 64379, 31680, 32806, 8665, 8581, 60429, 27189, 7004, 14490, 27959, 40120, 61965, 57446,
40767, 9506, 30011, 63676, 16650, 6053, 6459, 19781, 26735, 50643, 19042, 51938, 16298, 19033, 7838,
43301, 51725, 57656, 63232, 51368, 65031, 46605, 55392, 9509, 32286, 50079, 10218, 37000, 40932, 40890,
4415, 60392, 47891, 48141, 33494, 64130, 25837, 41840, 27717, 57910, 19235, 40414, 32108, 33204, 51667,
59269, 8221, 24221, 26572, 53731, 59583, 12556, 15280, 8670, 571, 20383, 25326, 13967, 13776, 30529,
2175, 7819, 8110, 64263, 16570, 35558, 55085, 12863, 21481, 53120, 54957, 30280, 2210, 16659, 52235,
27616, 42152, 33507, 29239, 51945, 32471, 47977, 8414, 10609, 27084, 2738, 40268, 8843, 59468, 27787,
56664, 61642, 25038, 39276, 9981, 21508, 17725, 64495, 7775, 63696, 2659, 17292, 27874, 55810, 35170,
19244, 13361, 40907, 13019, 21447, 16367, 34450, 54737, 54046, 65365, 28076, 49056, 61876, 4276, 8795,
26780, 24477, 43398, 35627, 18815, 24692, 8364, 39195, 29516, 33998, 9633, 32619, 21929, 3803, 58244,
49410, 45617, 9974, 49174, 8108, 19506, 4053, 12516, 502, 23668, 8665, 55033, 44229, 43647, 10663,
14877, 42960, 41370, 27958, 45473, 39233, 56912, 4757, 39967, 5703, 21297, 58081, 4677, 7777, 52981,
30204, 18837, 12346]
```

Il y a au moins un doublon : 8665. On a obtenu ce résultat en programmant un algorithme qui génère une liste de 302 entiers compris entre 1 et 2^{16}, puis un algorithme de recherche de doublon.
Voici, écrit de manière très libre, le code d'une fonction testant la présence de doublons dans une liste.

Algorithme

```
fonction doublon( contenu : liste ) -> booléen
    déjà_vu est vide
    pour x dans contenu
        si x est dans déjà_vu
            renvoyer vrai
        sinon
            ajouter x à déjà_vu
        renvoyer faux
```

On commence par créer une structure qu'on note `déjà_vu`, puis en parcourant la liste `contenu`, on ajoute à `déjà_vu` les éléments en vérifiant qu'il n'y sont pas déjà. Si c'est le cas, on s'arrête.
Pour programmer cet algorithme, il nous faut créer une structure de donnée appropriée

pour `déjà_vu`. Laquelle ? Cela dépend : on peut vouloir un algorithme très rapide ou qui ne consomme pas trop de mémoire ou… d'autres choses encore.

Pour la suite, on décide d'appeler `s` la structure de donnée qui va représenter `déjà_vu`.

1 Le choix le plus simple : avec une liste

On décide que `S` est une liste toute simple. Voici ce que l'on obtient :

Code Python

```python
def has_duplicates_1(content: list) -> bool:
    s = []
    for x in content:
        if x in s:
            return True
        else:
            s.append(x)
    return False
```

L'avantage est que c'est très simple à écrire. Ceci dit lorsqu'on vérifie `if x in s`, ce test nécessite en gros autant d'opérations que la longueur de `s`, et on peut avoir envie de minimiser ces opérations en choisissant une autre structure de données pour `s`.

2 Avec une liste de booléens

On a 65 536 valeurs possibles pour chaque nombre de `content`. Donc on peut créer une liste `s` de 65 536 booléens valant `False` et, en parcourant `content`, mettre les indices de `s` à `True` comme ceci :

Code Python

```python
def has_duplicate2(content: list) -> bool:
    s = [False] * 65536
    for x in content:
        if s[x] == True:
            return True
        else:
            s[x] = True
    return False
```

C'est encore très simple et le test `if s[x] == True` ne prend pas beaucoup de temps puisque c'est un simple accès à un élément de `s`, et pas un parcours de `s` en entier.
Le problème est que PYTHON ne représente pas les booléens comme des bits en mémoire, mais comme des `int`, c'est-à dire qu'un booléen est stocké sur... 64 bits. Quel gaspillage de mémoire !
On peut donc essayer une autre méthode.

3 Avec les bits d'un grand nombre

On va faire la même chose que précédemment, mais au lieu de créer un tableau de 65 536 booléens, on va travailler sur un nombre à 65536 bits, valant 0 au départ et dont on met le bit numéro `x` à 1 quand on trouve le nombre `x` dans la liste `content` :

Code Python

```python
def has_duplicate3(content: list) -> bool:
    s = 0
    for x in content:
        v = 1 << x
        if s & v:
            return True
        else:
            s = s | v
    return False
```

En théorie cela marche bien étant donné que PYTHON gère des entiers arbitrairement grands. En pratique, nous verrons que c'est extrêmement lent.

4 Une solution hybride : la liste de paquets

Puisque `content` contient 302 nombres, on va créer une liste de taille 302, mais pas une liste de nombres, une liste de listes qu'on appelle liste de paquets, et on va s'arranger pour que chacun de ces paquets soit très petit en taille. Concrètement quand on trouve dans `content` un nombre `x` compris entre 0 et 65 535, on le range dans `s` dans le paquet d'indice `x%302` (on rappelle que `%` signifie modulo) :

Code Python

```python
def has_duplicate4(content: list) -> bool:
    s = [[] for _ in range(302)]
```

```
    for x in content:
        if x in s[x % 302]:
            return True
        else:
            s[x % 302].append(x)
    return False
```

C'est raisonnable en terme de mémoire et aussi en terme de calculs, puisque le test `if x in s[x \% 302]` porte sur un paquet (qui est une liste, rappelons-le) qui sera probablement de taille très réduite.

En définitive, ces 4 fonctions sont quatre *implémentations* destinées à résoudre le problème de départ.

Définition : implémentation

Une implémentation est un *choix concret* de programmation pour répondre à un problème qui, lui, peut s'avérer *abstrait* :

- choix des structures de données pour représenter les variables ;
- parfois, choix d'un algorithme particulier, ou d'un paradigme de programmation.

Dans le cadre de notre exemple, les 4 implémentations diffèrent par le choix d'une structure de données. Au chapitre « Récursivité », on a rencontré 2 implémentations de la fonction `factorielle` : l'une impérative, l'autre récursive.

II Modularité

On se rend compte que les 4 programmes précédents ont la même allure. D'ailleurs pour peu qu'on écrive 3 fonctions

- `create`, qui crée une structure de donnée vide ;
- `contains`, qui vérifie la structure contient déjà ou non un élément ;
- `add`, qui ajoute un élément à la structure.

On peut tous les résumer ainsi :

Code Python

```
def has_duplicate(content: list) -> bool:
    s = create()
    for x in content:
        if contains(s,x):
            return True
        else:
            add(s,x)
    return False
```

Notion de modularité

La modularité, c'est le fait d'écrire un programme en le découpant en plusieurs *composants*, ces composants peuvent être par exemple des fonctions ou des objets (on les définira plus tard).
Nous avons déjà évoqué l'intérêt de procéder ainsi pour les fonctions : chacune d'elle a sa propre *spécification* et fonctionne de la manière la plus indépendante possible.

L'étape suivante est de regrouper des fonctions dans un *module*, qui sera un simple fichier `.py`, que l'on pourra ensuite importer comme on a l'habitude de le faire. Ainsi on pourra :

- *réutiliser* cette fonction dans un autre programme sans avoir à la réécrire ;
- Si on se rend compte qu'on peut améliorer une fonction (temps de calcul, gain de mémoire), on pourra réécrire le corps de cette fonction sans modifier ses spécifications et les programmes qui l'utilisent fonctionneront toujours sans les modifier.

Voici un premier module qui illustre le principe : on a construit un module destiné à la recherche de doublons dans des listes de 302 nombres compris entre 1 et 65 536.
Le choix de la structure `s` est une simple liste, comme à la partie I.1.

duplicates1.py

Code Python

```
"""
This module helps the user find duplicates in a list of 302
    values which are integers ranging from 1 to 2**16
"""


def create() -> list:
    """
    creates the structure to store values
    :return: the empty structure
    """
    return []


def contains(l: list, x: int) -> bool:
    """
    Checks whether a value belongs to the structure or not
    :param l: the structure
    :param x: the integer
    :return: bool
    """
    if x in l:
        return True
    else:
        return False


def add(l: list, x: int) -> None:
    """
    adds a new value to the structure
    :param l: the structure
    :param x: the value
    :return: None
    """
    l.append(x)
```

Remarque

On a bien pris soin

- d'écrire une *docstring* au début du module pour expliquer ce qu'il fait ;
- de préciser la spécification de chaque fonction avec des indication de type et aussi avec une *docstring*.

De cette manière, dans une console PYTHON, tout utilisateur peut taper

```
>>> from duplicates1 import *
>>> help(duplicates1)
```

pour obtenir la documentation qu'on a incluse.

Le programme suivant illustre l'utilisation de notre module.

Code Python

```
from duplicates1 import *
from random import randint


def has_duplicate(content: list) -> bool:
    s = create()
    for x in content:
        if contains(s, x):
            return True
        else:
            add(s, x)
    return False


test_list = [randint(1, 2 ** 16) for _ in range(302)]

print(has_duplicate(test_list))
```

En moyenne environ une fois sur deux, il affiche `True` et une fois sur deux `False`.

Exercice 1

Le module `duplicates1` a été écrit en utilisant la fonction `has_duplicates1`.
Écrire les modules `duplicates2`, `duplicates3` et `duplicates4` et vérifier que le pro-

gramme précédent marche tout aussi bien en utilisant ces modules (utiliser PyCharm pour travailler simultanément sur plusieurs fichiers).

III Interface

Notion d'interface

L'interface d'un module ou d'une *structure de données* (on reparlera plus tard des structures de données), c'est l'ensemble des fonctions dont l'utilisateur du module (ou de la structure) dispose, avec leurs spécifications et de la documentation. Il peut aussi y avoir les variables que l'utilisateur peut modifier (nous allons voir comment ci-dessous) et aussi des « constantes » dont l'utilisateur peut se servir.

L'interface d'un module est finalement toute la partie *publique* du module, qui lorsqu'il est bien documenté, est décrite à l'aide de docstrings accessibles grâce à la fonction `help` de Python.
Pour notre exemple de module (`duplicates1` ou les 3 autres), l'interface est la même : ce sont les fonctions `create`, `contains` et `add`. Il n'y a pas de constantes ni de variables.

Remarque

L'interface décrit à l'utilisateur *comment utiliser* le module mais ne précise pas *comment il fonctionne*.
En d'autres termes, l'interface n'explique pas comment ses fonctionnalités sont implémentées.

IV Encapsulation

Notion d'encapsulation

L'encapsulation, c'est le fait que certaines variables (et fonctions) restent internes au module, cachées à l'utilisateur. On peut considérer que tout ce qui n'est pas accessible *via* l'interface du module est encapsulé.

En Python, l'utilisateur peut lire le code d'un module et accéder à toutes les variables et fonctions du module mais ce n'est pas conseillé : après tout, pourquoi utiliser autre chose que l'interface du module s'il est bien programmé et que tout fonctionne comme prévu ?
Ainsi on utilise parfois la convention suivante lors de l'écriture d'un module en Python :

toutes les variables et fonctions dont le nom commence par « _ » sont destinées à être privées.
Dans d'autres langages (C++, JAVA entre autres) on peut déclarer des variables *publiques* (accessibles par l'utilisateur) ou *privées*.

Voici à quoi peut ressembler un module en PYTHON :

Code Python

```
"""
This module does this and that.
"""

_total = [0]


def this(a: int) -> None:
    """
    does this
    :param a: int
    :return: None
    """
    _total[0] += a
    _display()


def that(a: int) -> None:
    """
    does that
    :param a: int
    :return: None
    """
    _total[0] -= a
    _display()


def _display():
    print(f'total value is :', _total[0])
```

Exercice 2

Sans écrire et exécuter ce module

1. Expliquer ce qu'il fait.
2. Quelle est son interface ?
3. Quelles sont les variables et fonctions privées ?
4. Pourquoi selon vous, n'a-t-on pas utilisé une variable `_total` de type `int` ?

Projet : module `fractions_custom`

On a commencé à écrire l'interface d'un module nommé `fractions_custom` (le module `fraction` existe déjà). Son but est de travailler avec des *fractions*. La voici :

```
def create(a: int, b: int):
    """creates a/b"""
    pass

def add(f1, f2):
    """adds f1 and f2"""
    pass

def sub(f1, f2):
    """subtracts f2 from f1"""
    pass

def mul(f1, f2):
    """multiplies f1 and f2"""
    pass

def div(f1, f2):
    """divides f1 by f2"""
    pass

def display(f):
    """prints f on screen"""
    pass
```

Pour l'exercice, la documentation est très limitée car c'est à toi d'écrire le module.
en suivant la démarche suivante :

– choisir une structure de donnée pour représenter une fraction ;

– implémenter toutes les fonctions de l'interface en fonction du choix.

Pour l'instant on ne demande **aucune gestion des erreurs** : si l'utilisateur veut créer $\frac{14}{0}$, le module renverra peut-être une erreur ou fera n'importe quoi mais ce n'est pas le problème (de toutes façons les gens qui effectuent des divisions par zéro ne devraient pas avoir le droit d'utiliser un module PYTHON).La gestion des erreurs sera traitée dans un prochain chapitre.

Pour aller plus loin, on pourra simplifier systématiquement les fractions en écrivant une fonction `_simplify` ne faisant pas partie de l'interface (et pour simplifier $\frac{a}{b}$ il suffit de diviser a et b par le pgcd de ces deux nombres, fonction qu'on a déjà rencontrée.)

Projet optionnel : module `kb_mouse`

En utilisant les fonctions keybd_event, mouse_event et les constantes VK_Codes de l'API Win 32 diponibles avec PyWin32, créer un module nommé `kb_mouse` qui permet (entre autres) de :

– vérifier si une touche du clavier ou un bouton de la souris est en train d'être pressée ;

– simuler un clic de souris ou la pression sur une touche du clavier ;

– simuler un clic de souris à un endroit précis de l'écran ;

– (plus dur) bouger de manière continue d'un point à un autre de l'écran pendant un intervalle de temps donné.

On pourra utiliser les fonctions `sleep` et `perf_counter` du module `time` pour le dernier point.