Lorsqu'on développe un logiciel à grande échelle, une des clés consiste à bien séparer les différentes parties du programme. On peut par exemple :

— séparer une structure de données (ensemble de fonctions, classes, etc.) du code qui les utilise ;

— séparer l'interface utilisateur du coeur de l'application.

Pour cela on place les différentes parties du code dans différents fichiers appelés modules.

Remarque : On connaît déjà des modules préexistants en Python (`math`, `random`, `tkinter`, etc.). Ici, on détaille comment créer et utiliser ses propres modules.

1 Un exemple d'utilisation de la modularité

On se propose de répondre au problème de la recherche d'un doublon dans un tableau contenant des dates d'anniversaire afin d'illustrer la paradoxe des anniversaires :

> Dans un groupe contenant 23 ou plus, on a plus de 50% de chance d'avoir un anniversaire commun.

Une première possibilité consiste à utiliser la structure `set` de Python :

```
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
```

Le problème de cette solution est que, à moins de bien connaître la façon dont les ensembles sont implémentés en Python, on ne sait pas vraiment comme l'ajout dans l'ensemble est géré, et notamment son coût. Modifions le programme pour que celui ci-fonctionne sans la structure `set` :

```
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
```

La fonction a changé mais on connaît maintenant le coût de notre recherche : puisqu'on parcourt les tableaux `t` et `s`, on a une complexité quadratique dans le pire cas (aucun doublon).

Il est ici cependant possible de faire mieux dans le contexte de notre problème : puisque les éléments testés sont des dates, elles sont en nombre fini (disons un nombre entre 1 et 366). Ainsi, il est possible de garder en mémoire non pas un tableau contenant les éléments déjà rencontrés mais plutôt un tableau de 367 cases (jour 0 inexistant jusque jour 366) contenant des booléens disant si un jour a déjà été rencontré dans la recherche.

```
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = [False]*367
    for x in t:
        if s[x]:
            return True
        s[x]=True
    return False
```

L'avantage de cette méthode est que sa complexité temporelle est linéaire en la taille de `t`. Cependant, on crée et garde en mémoire un tableau de taille 367 alors que l'on va tester 23 dates... On peut cependant combiner les avantage des deux méthodes en utilisant une structure appelée tableau de hachage :

```
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = [[] for i in range(23)]
    for x in t:
        if x in s[x%23]:
            return True
        s[x%23].append(x)
    return False
```

Dans cette solution, on crée un tableau de taille 23 contenant des sous-tableaux de dates déjà rencontrées. Si on en croit le paradoxe des anniversaires, au bout de 23 éléments, on a plus d'une chance sur deux d'avoir un doublon et donc de le détecter. On stocke donc les dates selon une **fonction de hachage** (`x%23`) assurant, si les dates sont aléatoirement réparties, que chaque sous tableau ne sera pas trop grand.

On peut encore faire mieux mais arrêtons nous un instant. Si on reprend les fonctions écrites jusque maintenant, on se rend compte qu'on a écrit 4 fois à peu prêt le même programme :

```
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = <definission d une structure>
    for x in t:
        if <x deja rencontre>:
            return True
        <ajouter x a la structure>
```

Ce qu'on pourrait écrire, en définissant les bonnes fonctions `cree()`, `contient(s,x)` et `ajoute(s,x)` :

```
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)
```

Si on décide d'écrire correctement ces trois fonctions, on sait que notre fonction `contient_doublon(t)` va marcher et mieux : on pourra toujours les améliorer sans réécrire entièrement le programme principal. Par "écrire correctement", on entend que :

1. `cree()` initialise une structure vide allant contenir des dates ;
2. `contient(s,x)` teste si la date `x` est dans la structure `s` créée par la fonction `cree()` ;
3. `ajoute(s,x)` ajoute la date `x` à la structure `s` créée par la fonction `cree()`.

Mieux, il est même possible de mettre ses trois fonction dans un fichier séparé de notre programme principal si ces trois conditions sont vérifiées.

On dit que les conditions 1, 2 et 3 forment **une interface** et que le fichier contenant les fonctions (si elles vérifient 1, 2, et 3) est un `module` qui **réalise** cette interface

Un exemple de réalisation du module `date` :

```
#implementation de la structure de dates, version 3.0

def cree():
"""initiallise une table de hachage pour stocke des dates (nombres entiers entre 1 et 366)"""
    return [[] for i in range(23)]

def contient(s,x):
"""teste si un entier x est dans la table de hachage s"""
    return x in s[x%23]

def ajoute(s,x):
"""ajoute un entier x entre 1 et 366 dans la table de hachage s"""
    s[x%23].append(x)
```

2 Modules, interface et réalisation

Un module est un fichier contenant un ensemble de fonctions outils pouvant être utilisées dans un logiciel plus large.

Pour créer un module en Python, il suffit de regrouper ces fonctions dans un unique fichier :

`<nom du module>.py`

Pour importer un module personnel dans un programme, on procède comme avec les modules préexistant en Python :

- `import <nom du module>`
- `import <nom du module as <nom raccourci>`
- `from <nom du module> import <fct1>, <fct2>, <etc.>`
- `from <nom du module> import *`

Pour chaque module on distingue :

— sa réalisation (ou implémentation), c'est à dire le code lui même ;

— son interface, consistant en la liste des fonctions utilisables du module, assorties d'une description de celles-ci.

Une interface est un moyen pour un utilisateur du module de pouvoir prendre en main celui-ci sans devoir comprendre comment il fonctionne. Pour cela il est important d'être clair dans la description des fonctions.

On doit y préciser :

— l'effet des différentes fonctions ;

— les arguments principaux et optionnels ;

— leurs types et éventuellement les valeurs interdites.

Comme on l'a vu dans l'exemple introductif, une même interface peut avoir plusieurs réalisations différentes, c'est même un des intérêts de la modularité.

Une interface est un contrat entre le concepteur du module et son utilisateur. Le concepteur assure à travers l'interface que, si les conditions d'utilisation des fonctions du module sont respectées, celles-ci auront l'effet promis dans l'interface. L'utilisateur n'a d'ailleurs pas besoin de connaître le détail de la réalisation.

3 Encapsulation

Lors de la réalisation d'un module, il peut arriver que l'on ait besoin de un certain nombre de fonctions auxiliaires ou de variables globales ne relevant pas de l'interface du module elle même. Ce sont des fonctions et des variables relevant de la machinerie du module et qui n'ont pas vocation à être utilisées par l'utilisateur.

Ces objets internes ne figurent évidemment pas dans l'interface mais on pourrait aller plus loin en interdisant à l'utilisateur d'y avoir accès : on parle alors d'encapsulation du code.

En Python il est malheureusement impossible d'interdire l'accès aux objets définis dans un module mais il existe une convention consistant les nommer en commençant par un underscore `'_'`.

Par exemple, dans notre module `date`, on pourrait définir une variable une variable `taille` fixant la taille de la table de hachage. Cette variable relève de la réalisation du module uniquement et ne doit pas être utilisée par l'utilisateur. On la nomme donc ici `_taille` :

```
#implementation de la structure de dates, version 3.1

_taille = 23

def cree():
"""initiallise une table de hachage pour stocke des dates (nombres entiers entre 1 et 366)"""
    return [[] for i in range(_taille)]

def contient(s,x):
"""teste si un entier x est dans la table de hachage s"""
    return x in s[x%_taille]

def ajoute(s,x):
"""ajoute un entier x entre 1 et 366 dans la table de hachage s"""
    s[x%_taille].append(x)
```

4 Exceptions

Pour l'utilisateur qui ne connaît pas le détail de l'implémentation des fonctions d'un module, il est important de signaler clairement les erreurs lorsqu'elles surviennent. On connaît bien l'utilisation de la commande `assert` en Python mais en pratique, on la réserve pour les tests internes au développement.

Lorsque l'on veut signaler des erreurs à un utilisateur, on utile plutôt des erreurs ou plutôt exceptions. Il existe plusieurs type d'exceptions :

Exception	Contexte
`NameError`	Accès à une variable inexistante
`IndexError`	Accès à un indice invalide de tableau
`KeyError`	accès à une clé inexistante d'un dictionnaire
`ZeroDivisionError`	division par zéro
`TypeError`	opération appliquée à des valeurs impossibles
`ValueError`	opération appliquée à des valeurs impossibles

Pour l'utilisateur qui ne connaît pas le détail de l'implémentation des fonctions d'un module, il est important de signaler clairement les erreurs lorsqu'elles surviennent, notamment les erreurs liées au non respect des conditions de l'interface. On connaît bien l'utilisation de la commande `assert` en Python mais en pratique, on la réserve pour les tests internes au développement.

Lorsque l'on veut signaler des erreurs à un utilisateur, on utile plutôt des exceptions. Il existe plusieurs type d'exceptions :

Exception	Contexte
`NameError`	Accès à une variable inexistante
`IndexError`	Accès à un indice invalide de tableau
`KeyError`	accès à une clé inexistante d'un dictionnaire
`ZeroDivisionError`	division par zéro
`TypeError`	opération appliquée à des valeurs impossibles
`ValueError`	opération appliquée à des valeurs impossibles

Lever une exception : Il est possible de déclencher manuellement n'importe quelle exception via la commande `raise` :

```
raise <nom de l'exception>(<texte à afficher>)
```

Prenons par exemple la fonction `ajoute` du module dates. On peut lever une exception lorsqu'on appelle sur un entier non compris entre 1 et 366 :

```
def ajoute(s,x):
"""ajoute un entier x entre 1 et 366 dans la table de hachage s"""
    if not(1<=x<=366):
        raise ValueError("Date " + str(x) + " invalide.")
    s[x%_taille].append(x)
```

Remarque : On n'a pas besoin de `else` ici car l'exception interrompt l'exécution du programme si elle survient.

Il peut aussi être utile de signaler manuellement des exceptions étant déjà levées automatiquement afin de respecter l'encapsulation du code. Par exemple, une exception `IndexError` dans une fonction d'un module utilisant un tableau va renseigner l'utilisateur sur la réalisation de celui-ci, ce qui n'est pas nécessaire.

Parfois, il est possible de ne pas vouloir interrompre l'exécution du programme en cas d'erreur. On parle alors de rattrapage d'exception.

Rattraper une exception : Il est possible de rattraper une exception à l'aide de la syntaxe `try` / `except` :

```
try:
    <bloc d instructions protege>
except <nom de l exception a rattraper 1>:
    <bloc d instructions alternatif 1>
except <nom de l exception a rattraper 2>:
    <bloc d instructions alternatif 2>
...
```

Il est aussi possible de rattraper n'importe quelle exception si on ne précise rien après le `except` (déconseillé) :

```
try:
    <bloc d instructions protege de toute exception>
except:
    <bloc d instructions alternatif>
```

Cela est notamment utile pour remplir un fichier de journalisation (log), c'est à dire un fichier gardant en mémoire toutes les erreurs survenant lors de l'exécution d'un programme.

```
def ajoute(s,x):
"""ajoute un entier x entre 1 et 366 dans la table de hachage s"""
    try:
        if not(1 <= x <= 366):
            raise ValueError
        s[x%_taille].append(x)
    except ValueError:
        f = open("log.txt", 'a')
        f.write(ctime() + ", module time, fonction ajoute : date " + str(x) + "invalide.")
        f.close()
```

Ici, en cas d'erreur et plutôt que d'afficher un message dans le terminal, on renseigne dans un fichier texte appelé `log.txt` la date et l'heure auxquelles l'erreur est survenue ainsi que la nature de l'erreur.

Remarque : La fonction `ctime` est une fonction du module `time` de Python, à importer dans le module `dates`.