Abstraction et interface

Exemple d'introduction

Ada et Linus sont deux étudiants et ils travaillent sur un projet permettant de calculer des probabilités. Pour cela, ils veulent passer par des fractions, plutôt que des réels, à cause des erreurs de virgules flottantes. Ils ont besoin de décider comment représenter les fractions. Ils hésitent entre un couple d'entiers, un tableau avec deux entiers et un dictionnaire :

```
>>> f = (1, 3)
>>> f[0]  # numérateur
1
>>> f[1]  # dénominateur
3
```

```
>>> f = [1, 3]
>>> f[0]  # numérateur
1
>>> f[1]  # dénominateur
3
```

```
>>> f = {'numerateur': 1, 'denominateur': 3}
>>> f['numerateur']
1
>>> f['denominateur']
3
```

Le tableau et le dictionnaire sont mutables mais pas le couple. En dehors de cela, il n'y a pas de grosse différence entre le couple et le tableau. Le dictionnaire permettrait de rajouter des informations supplémentaires, comme par exemple la valeur approchée. Les deux étudiants hésitent, alors Ada se propose de s'en occuper pendant que Linus commence à implémenter des fonctions pour les opérations de base.
Mais comment faire pour obtenir le numérateur et le dénominateur d'une fraction s'il ne sait pas si elle sera représentée par un couple ou un dictionnaire ? Ada lui propose alors de faire 3 fonctions permettant de créer une fraction, d'obtenir son numérateur et son dénominateur :

```
>>> f = fraction(1, 3)
>>> numerateur(f)
1
>>> denominateur(f)
3
```

Ainsi, Linus peut travailler sans que ses fonctions ne dépendent des choix d'Ada. De son côté, elle peut faire une première implémentation très simple, par exemple avec un couple et si elle décide de passer sur un dictionnaire, elle n'aura que ces 3 fonctions à modifier et cela ne changera rien pour les autres fonctions rajoutées après.

Abstraction et interface

Cet exemple montre le principe de l'abstraction des structures de données et des interfaces. Le module qu'ils vont proposer permettra de manipuler des fractions sans avoir à connaître leur implémentation en interne. C'est l'**abstraction** des structures de données. La description des fonctions à la disposition des utilisateurs s'appelle l'**interface**.
L'intérêt pour l'utilisateur, c'est qu'il n'est pas nécessaire de connaître les détails internes, ni le code des fonctions. Lorsque vous utilisez la tortue Python, vous n'avez pas besoin de savoir comment elle est représentée en mémoire ou comment est programmée la fonction `forward`. Vous avez juste besoin de savoir qu'elle existe et à quoi elle sert.

Améliorer le module

Pour l'instant l'interface est :

Fonction	Description
`fraction(n, d)`	Renvoie une fraction correspondant à `n/d`, avec `d` non nul.
`numerateur(f)`	Renvoie le numérateur de la fraction `f`.
`denominateur(f)`	Renvoie le dénominateur de la fraction `f`.

Linus travaille sur la fonction permettant d'additionner deux fractions :

```
def somme(f1, f2):
    n1 = numerateur(f1)
    d1 = denominateur(f1)
    n2 = numerateur(f2)
    d2 = denominateur(f2)
    return fraction(n1*d2 + n2*d1, d1*d2)
```

Le problème, c'est qu'aucune simplification n'est faite et il aimerait bien que `fraction` renvoie une fraction réduite et telle que le dénominateur est positif. Il décide dont de rajouter une fonction `simplifier(f)` qui permet de réduire une fraction. Il hésite entre deux approches pour faire cette fonction : soit chercher et utiliser tous les diviseurs communs du numérateur et du dénominateur, soit calculer directement le PGCD des deux nombres. Là encore, peu importe son choix, Ada pourra utiliser `simplifier(f)` avant de renvoyer de résultat dans `fraction(n, d)`.
L'interface devient alors :

Fonction	Description
`fraction(n, d)`	Renvoie une fraction irreductible correspondant à `n/d` et dont le dénominateur est positif et non nul.
`numerateur(f)`	Renvoie le numérateur de la fraction `f`.
`denominateur(f)`	Renvoie le dénominateur de la fraction `f`.
`somme(f1, f2)`	Renvoie une fraction irreductible égale à `f1` + `f2` et dont le dénominateur est positif.

La fonction `simplifier(f)` n'apparaît pas puisqu'elle est inutile pour l'utilisateur, vu que toutes les fractions renvoyées par les autres fonctions sont déjà réduites. On parle alors de fonction **privée**, en opposition aux fonctions **publiques** qui sont présentées dans l'interface. De la même manière, si Linus décide de faire une fonction pour calculer le PGCD, elle n'a pas à apparaître dans l'interface. Surtout s'il décide de l'enlever dans une prochaine version du programme. Par convention, on met un "_" devant les noms des fonctions ou constantes privées. Ainsi, Linus appellera sa fonction `_simplifier(f)`. Cela ne change rien pour Python, c'est uniquement une convention entre développeurs.

Intérêt de la modularité

Le découpage en plusieurs modules d'un projet permet de se répartir la tâche à plusieurs. Il permet aussi de s'assurer qu'un changement de structure de données sur une des parties ne nécessite pas de modifier un grand nombre de fonctions.
En séparant l'interface de l'implémentation, on augmente la modularité du programme et on facilite la recherche et la correction de bugs, ainsi que l'ajout de nouvelles fonctionnalités. Mais cela nécessite aussi de bien définir à l'avance les interfaces des principales fonctions et de bien documenter le code pour savoir clairement à quoi sert chaque fonction, et idéalement, comment elle fonctionne.