Modules, Interfaces et Encapsulation

1. Retour sur les modules

Un module python est un fichier python contenant des fonctions, des constantes (et des constructeurs d'objets, mais nous verrons ça un peu plus tard) regroupées dans ce fichier, car elles traitent de la même structure de données.

📋 Un exemple

Par exemple, dans un logiciel de géométrie dynamique tel que GeoGebra, on pourrait avoir :

- un module qui s'occupera exclusivement de la partie graphique, c'est-à-dire du tracé des formes ;
- un module qui s'occupera des interactions avec l'utilisateurs ce qu'on appelle la **GUI** (*Graphic User Interface*) :
- un ou des modules pour calculer les intersections entre des formes géométriques ;
- bien d'autres possibilités...
- et le programme principal qui lui importera les modules précédents et gèrera les évènements liant tous les objets mathématiques et les évènements de l'utilisateurs. On dira que ce programme dépend des autres modules.

Importer un module

Il existe plusieurs possibilités pour importer un module. Dans la suite des exemples, le module toto, contenant les fonctions bidule() et truc() est situé:

- soit dans le même dossier que le fichier qui l'importe;
- soit dans un dossier accessible par le système (dossier faisant partie de la variable PATH de windows par exemple).

Import complet

```
import toto
toto.bidule()
toto.truc()
```

Toutes les fonctions du module toto sont importées, et elles le sont dans un namespace (espace de nommage) spécifique. Les fonctions sont alors appelées en les préfixant par le nom du module (toto.).

Import complet avec alias

```
import toto as to
to.bidule()
to.truc()
```

Comme pour l'import complet, toutes les fonctions du module toto sont importées, et elles le sont dans un namespace (espace de nommage) spécifique. Les fonctions sont alors appelées en les préfixant par l'alias du module (to1.).

Import partiel dans le namespace courant

```
from toto import bidule
bidule()
```

lci, seul la fonction bidule() est importée, mais elle l'est directement dans le namespace principal (autrement dit main) du fichier effectuant l'import. La fonction truc() n'est pas appelable (elle n'existe pas pour l'interpréteur).

Import complet dans le namespace courant

```
from toto import *
bidule()
truc()
```

Toutes les fonctions sont appelées directement dans l'espace de nom principal (soit le namespace de main) du fichier effectuant l'import.

⚠ C'est une pratique périlleuse! Si une fonction du module porte le même nom qu'une fonction du fichier appelant, celle importée écrasera celle du fichier courant. C'est parfois utile lorsqu'on veut modifier le comportement d'une fonction(voir l'exemple ci-dessous), mais attention on peut alors rencontrer des erreurs majeures!

Fichier principal

```
from mon_module import *
print("Pain ")
```

Fichier mon_module.py

```
def print(truc) :
    print(truc*3+"Tarte Tatin"*2)
```

Sortie attendue

Probablement, ce qui est attendu par la sortie du fichier principal est juste la chaîne de caractères "Pain ". Mais la fonction **built-in** print() a été écrasée par celle du module mon_module.py.

Sortie réelle

```
"Pain Pain Pain Tarte Tatin"
```

La véritable fonction appelée est celle du module mon_module.py.

(Pour ceux qui ont en tête la marche impériale, c'est normal...)

Remarque

Il est toutefois possible d'importer un fichier présent dans un autre dossier :

- soit en utilisant une adresse absolue import 'C:\Mes_Programmes\Python\toto.py';
- soit en utilisant une adresse relative import '..\toto.py' (si toto.py est situé dans le dossier parent du fichier qui importe).

2 Interfaces

E Conception logicielle

Dans la conception de logiciels à grande échelle, ou à plusieurs programmeur·euse·s (et à l'heure actuelle il est fréquent d'avoir plusieurs milliers de programmeur·euse·s concevant un logiciel), il est important de pouvoir séparer les différents éléments du programme en sous-ensembles cohérents et ayant le minimum d'interactions entre eux. En particulier, ils se doivent d'être le plus étanches possibles quant à leur fonctionnement - c'est-à-dire les plus indépendants possible. On retrouve ici que la notion d'interface est essentielle.

Pour chaque module, on peut donc distinguer :

- une interface, c'est-à-dire la description des différentes fonctions du module et de leurs arguments (obligatoires ou facultatifs). Il s'agit donc d'une documentation la plus claire possible sur la manière d'utiliser le module.
- une **implémentation**, c'est-à-dire la manière dont sont codées ces fonctions (choix de structures, nom des variables intermédiaires, etc).

📋 Un exemple d'interface

Un module utilisable pour la fonction factorisée contient_doublon(t) aura pour interface:

fonction	Description
cree()	crée et renvoie un ensemble de date vide
<pre>contient(data,s)</pre>	renvoie True si la structure s contient la donnée data
ajoute(data,s)	ajoute la donnée data à la structure s

Vous constaterez que dans cette description, il n'est nul part fait mention de la *nature de la structure*. Il pourrait s'agir aussi bien de liste, de tableau de bits, de tables de hachage...

Exercice : réalisation de modules

Dans chacun des cas suivants, construire un module réalisant l'interface ci-dessus, et le tester en l'important dans le fichier rechercheDates.py où vous aurez modifié la fonction contient_doublon(t) par la version factorisée de celle-ci.

- 1. un module dateTab, dont la structure est implémentée sous la forme d'un tableau.
- 2. un module dateBool, dont la structure est implémentée sous la forme d'un tableau de booléen.
- 3. un module dateHash, dont la structure est implémentée sous la forme d'une table de hachage.
- 🧩 Il est bien entendu essentiel de s'inspirer des exemples donné dans la page d'introduction.

3. Notions d'encapsulation

Motion d'encapsulation

Le contrat qu'une interface établit entre l'utilisateur trice et l'auteur trice d'un module ne porte pas sur les moyens, mais sur les résultats : l'auteur trice s'engage à ce que les résultats produits par l'utilisation de ses fonctions soient bien ceux décrits dans l'interface, mais il ou elle est libre de s'y prendre comme il ou elle le souhaite.

En particulier il ou elle est libre d'introduire des fonctions, variables, constantes, ..., qui ne sont pas incluses dans l'interface. On parle alors de fonctions, variables, constantes encapsulées dans le module.

Le contrat explicite est que l'utilisateur trice <mark>ne doit en aucun cas</mark> utiliser ces données encapsulées. Dans le cas contraire, si l'auteur trice du module change son approche et modifie ces données internes, le programme du ou de la client·e risque de devenir non fonctionnel.

Norme en Python

En Python, l'auteur · trice d'un module peut indiquer que certains éléments sont privés (c'est-à-dire encapsulées) en faisant commencer leur nom par un caractère underscore .



Imaginons un module secondDegre.py dont l'interface est définie ainsi:

fonction	Description
polynome(t)	Vérifie que le tuple t sous la forme (a,b,c) représente bien un polynôme de degré 2, et renvoie t dans ce cas, et None sinon
valeursRacines (p)	Renvoie les valeurs des racines, et None si il n'existe pas de racines réelles
convexite(p)	Renvoie la convexité de la courbe représentative du polynôme sous la forme d'une chaîne de caractère en minuscule
tangente(p,x0)	Renvoie l'équation de la tangente à la courbe du polynôme p en x0

Dans l'interface de ce module, on considère que le calcul du discriminant est une opération privée.

On aurait alors comme possibilité d'implémentation (non complète):

```
from math import sqrt
def polynome(t) :
   a, b, c = t
    if a == 0 :
       return None
    return t
def _discriminant(p) :
    if polynome(p) is not None :
       a,b,c = polynome(p)
       return b**2 - 4*a*c
    return None
def _nombreRacines(p) :
def valeursracines(p) :
def convexite(p) :
def _calcule(p,x) :
def _nombreDerive(p, x0) :
def tangente(p, x0):
```

Dans ce module, les fonctions préfixées par 📃 sont considérées comme privées, et ne faisant pas partie de l'interface.



Exercice

Créer un module secondDegre.py contenant a minima la totalité des fonctions précédentes et implémenter toutes ces fonctions.

Encapsulation dans d'autres langages

Il faut noter que la notion de fonction ou variable privée en Python n'est qu'une convention. Rien $n'emp\ \hat{e} che\ r\'{e}ellement\ l'utilisateur\cdot trice\ du\ module\ d'utiliser\ ces\ fonctions\ priv\'{e}es.$

C'est loin d'être le cas dans d'autres langages (comme C++ ou Java), qui introduisent un contrôle strict de l'encapsulation en rendant l'accès aux éléments privés impossible.