Parcours et méthodes

Tle

Spécialité NSI

Numérique et sciences informatiques

LES ÉTAPES POUR RÉUSSIR

- Cours complet
- Méthodes appliquées
- Exercices et QCM corrigés

ellipses

Cours complet

1 Interface et implémentation

1.1 L'implémentation

Un type abstrait de données est un type de données, ce qui inclut les méthodes pour agir sur ces dernières, **indépendant** de sa réalisation matérielle, que l'utilisateur n'a pas à connaître.
Imaginons, par exemple, ce que pourrait être un type `agenda`. Une donnée se composerait d'une `date`, d'`un horaire de début`, d'un `horaire de fin`, d'un `intitulé`. À ceci il faudrait ajouter certaines fonctions :

- `creerAgenda()` pour créer un nouvel agenda, au départ vide,
- `ajouterDonnée(data)` pour ajouter la donnée `data` à l'agenda,
- `fusionnerAgenda(ag2)` pour fusionner un nouvel agenda `ag2` avec l'agenda courant,
- etc.

Ces fonctions seraient le moyen exclusif d'interagir avec un `agenda`. Ceci afin d'assurer plus de contrôle et de simplifier la programmation.

Il est possible de réaliser concrètement le type agenda de bien des manières. Il faudra faire des choix :

- Les données sont-elles totalement ou en partie en mémoire vive ?
- Si elles sont en mémoire vive, où et comment sont-elles inscrites ? En un seul bloc ? En îlots discrets ? Dispersées ?
- Quelles sont les tailles en octets des emplacements réservés pour les dates ? Pour les intitulés ?
- Si on réalise cette implantation à l'aide d'un langage de programmation, lequel va-t-on utiliser ?
- Faut-il prévoir de l'espace vide pour l'accroissement en taille d'un agenda ? Comment ?

- Doit-on privilégier l'économie de place mémoire, la rapidité d'accès, la simplicité d'utilisation ?
- Va-t-on se servir d'un serveur distant pour le stockage persistant ? D'un serveur de base de données ?
- Etc.

Évidemment les éléments précédents ne sont pas indépendants. Par exemple, si l'on choisit de travailler avec le langage Python, on ne pourra pas agir directement sur la gestion de la mémoire comme on pourrait le faire en C ou en assembleur. Ce choix de langage induira ceux qui concernent la place mémoire et la rapidité d'accès.
Précisons quelques décisions essentielles dans l'implémentation d'une structure de données. Entre autres :

- taille fixe ou taille **dynamique** de l'ensemble de la structure ⇝ réservation d'espaces suffisants dans la mémoire,
- accès direct ou pas à chacune des données ⇝ nombre d'adresses mémoire caractérisant la structure,
- donnée d'un type unique ou **polymorphisme**. Si l'on reprend l'exemple de l'agenda : on peut vouloir que les dates soient **toujours** des triplets d'entiers : « (16,06,1959) » ou, au contraire, tolérer qu'elles puissent avoir différents formats comme « 16 juin 1959 », « [16/06/59] », « en 1959, le 16 juin », etc. On parle de polymorphisme dans ce dernier cas. ⇝ taille en mémoire de chaque donnée : fixe, majorée ou dynamique,
- données **mutables** ou pas ⇝ types d'accès à la mémoire.

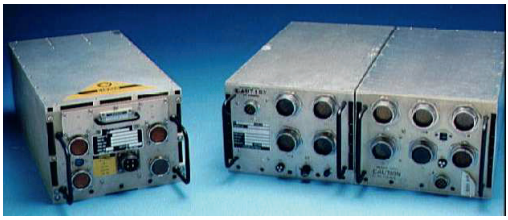
1.2 L'interface

L'interface d'une structure de données définit les modalités d'interaction avec cette structure. Elle est la seule face de cette dernière, visible pour l'utilisateur, que ce dernier soit une personne, un programme, une fonction ou un système d'exploitation.
Elle peut être riche, avec de nombreuses méthodes permettant toutes sortes de manipulations sur les données, ou austère, les méthodes étant, en nombre et en raffinement, réduites au strict minimum (c'est l'idée de la brique Lego avec laquelle on peut tout faire). Cela dépendra des ressources informatiques disponibles mais aussi du travail que l'on voudra déléguer à l'utilisateur.
On peut comparer l'interface d'une structure à la face avant d'un appareil électronique. Lorsqu'on se sert de ce dernier on ne connaît en général que cette face avant. Elle définit les actions possibles que l'on peut réaliser. Cette face avant peut comporter un grand nombre de commutateurs, boutons, cadrans, etc. , ou, au contraire, ne donner qu'un nombre restreint de contrôles.

2 Programmation objet

2.1 La Programmation modulaire

La navette spatiale américaine ne pouvait pas être pilotée par un être humain lors de son retour sur terre. Ce sont des ordinateurs (voir ci-dessous) qui étaient chargés de cette tâche complexe qui exigeait des prises de décision très rapides.



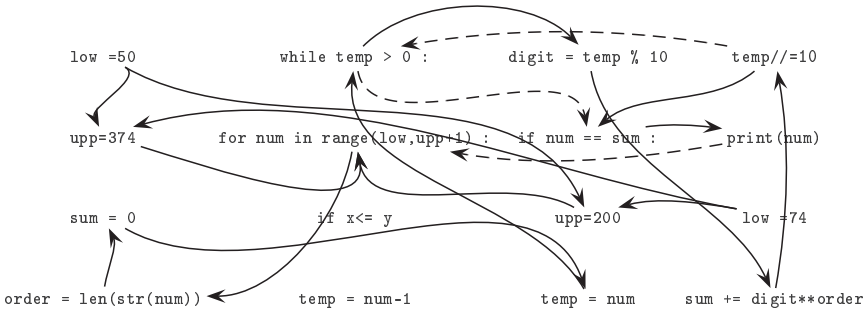
Des ingénieurs ont écrit le programme de ce retour sur terre. On peut remarquer :

1. sa dimension : il comporte environ 400000 lignes de code,
2. la multiplicité de ses auteurs : il a été écrit et réécrit par de nombreuses personnes, à des moments et des lieux différents.

Pour effectuer un tel travail, on utilise la programmation modulaire : celle-ci décompose un gros programme en morceaux, appelés modules, afin de pouvoir les écrire et les améliorer indépendamment. Le travail peut donc être éclaté dans le temps : sur plusieurs mois ou plusieurs années, et dans l'espace : attribué à des groupes de personnes indépendants.
Supposons qu'un des appareils de vol de la navette, disons A, soit remplacé par un autre plus fiable ou plus performant. Si le programme était d'une seule pièce alors il faudrait le réécrire en totalité ! Avec la programmation modulaire on ne doit revenir que sur le ou les modules où l'on traite de A.
De même si n'importe où, au beau milieu d'un module quelconque, se trouvait un ou des liens vers le module qui gère A alors la tâche serait vite insurmontable. On décide donc que chaque module est, pour les autres modules, une sorte de boîte noire, à laquelle les accès, lecture, écriture, effacement, etc. , ne peuvent se faire que par un unique « canal ». On appelle ce dernier l'interface du module (voir chapitre 1, section 1.2).
On peut très bien utiliser une automobile sans ouvrir le capot et démonter le moteur. Le conducteur ne connaît, en général, que l'interface, c'est-à-dire : le tableau de bord, le volant, les pédales et quelques accessoires.
Un module peut être remplacé par un autre sans changer quoi que ce soit aux autres modules, du moment que le remplaçant expose la même interface que celui qu'il remplace.
Bien sûr un programmeur maladroit travaillant sur un module X risquerait de faire un appel vers un élément d'un autre module Y sans passer par son interface. Pour éviter cela les variables et les fonctions internes d'un module sont **encapsulées**. Cela signifie qu'on ne peut y accéder que par l'interface et selon les modalités définies dans celle-ci.

2.2 Le code « spaghetti »

On appelle code « spaghetti » un code non structuré, dans lequel toute ligne de programme peut renvoyer à toute autre. C'est l'opposé de la programmation modulaire et de la programmation objet : tout est a priori connecté avec tout, n'importe quel élément du code peut dépendre de n'importe quel autre élément. Il saute aux yeux que ce type de programmation est très vite impossible à gérer.



2.3 La programmation objet

La programmation objet reprend et prolonge la programmation modulaire. Elle y ajoute d'autres concepts (seuls ceux qui appartiennent au programme de ce livre seront mentionnés).
On y définit des « briques » logicielles appelées objets ainsi que leurs interactions. Un objet possède une structure interne et un comportement qui régit ses interactions avec les autres objets. Il comporte ainsi :

- des données appelées attributs, c'est sa structure interne,
- des fonctions appelées méthodes qui sont les moyens d'interaction avec les autres objets.

Seuls certains attributs et certaines méthodes sont accessibles depuis l'extérieur de l'objet : on nomme ceci l'encapsulation. Les attributs et méthodes accessibles sans restriction depuis l'extérieur de l'objet sont dits « publics », les autres, pour lesquels le passage par une interface précisément définie est obligatoire, « privés ». Il existe, selon les langages, de nombreuses variations des droits d'accès aux objets. En Python, on distingue seulement « private » et « public ».
Les objets sont regroupés en classes : des objets de la même classe possèdent les mêmes attributs et les mêmes méthodes. La classe décrit la structure interne des données et définit les méthodes.
Dans un parallèle grossier, on peut dire que « Snoopy », « Rintintin », « Lassie » sont des objets de la classe « chiens ».

2.3.1 Les attributs

Prenons des exemples en Python pour illustrer certains concepts de base.

```
1 class Zbong  :
2     m = 45
3
4 x = Zbong()
5 print(x.m)
```

Les lignes 1 et 2 définissent la classe `Zbong`. On y trouve un attribut : `m`. Tout objet de classe `Zbong` aura donc cet attribut.
La ligne 4 est une instanciation de la classe `Zbong`. Cela signifie que l'on crée un objet, nommé `x`, de la classe `Zbong`. L'objet `x` possède donc l'attribut `m`, c'est ce que montre l'exécution : la ligne 5 résulte en l'affichage de « 45 ».

Poursuivons notre exemple :

```
1 class Zbong  :
2     m = 45
3
4 x = Zbong()
5 print(x.m)
6
7 x.m=100
8 print(x.m)
9 y = Zbong()
10 print(y.m)
```

La ligne 7 modifie l'attribut `m` de l'objet `x`. On vérifie à la ligne 8, qui affiche « 100 », le changement de valeur de l'attribut `m` de `x`.
À la ligne 9 on crée un deuxième objet de la classe `Zbong`, nommé, cette fois-ci, `y`. L'attribut `m` de `y` a bien la valeur 45 comme cela est écrit dans la définition de la classe `Zbong` ainsi que le montre l'exécution de la ligne 10.

Revenons sur la notation. On utilise pour marquer le lien entre un attribut et l'objet dont il dépend, un **point** :

Ainsi `x . m` désigne l'attribut `m` de l'objet `x`, `y . m` l'attribut `m` de l'objet `y` et `x . f()` la méthode `f()` de l'objet `x`.

```
1 class Zbong  :
2     m = 45
3
4 m = 88
5 x = Zbong()
6 print(x.m)
7 print(m)
```

Ligne 4 on définit une variable `m` à laquelle on donne la valeur 88. Cette variable n'est pas l'attribut `m` de l'objet `x`. La ligne 4 ne modifie pas la valeur de `x . m`. Ainsi les lignes 6 et 7 renvoient respectivement 45 et 88.

2.3.2 Les méthodes

```
1  class Zbong  :
2      m = 2
3      def f(self)  :
4          return '
               chat'
5      def g(self,t)
           :
6          return 3*t
7
8  x = Zbong()
9
10 print(x.f())
11 print(x.g(4))
12 print(x.g(x.m))
```

Ligne 7 on instancie un objet `x` de la classe `Zbong`. Cet objet possède deux méthodes : `f` et `g`.
Le mot clé `self` désigne l'objet instancié lui-même. Dans la définition d'une méthode c'est toujours le premier argument, comme on le voit lignes 3 et 5.
Lors de l'exécution d'une méthode, l'argument `self` est toujours passé **implicitement** en premier.
Cela signifie que les lignes 10, 11 et 12 sont respectivement exécutées comme `print(x.f(self))`, `print(x.g(self,4))` et `print(x.g(self,x.m))`.

2.3.3 L'encapsulation

Dans Python, par défaut, tous les attributs et toutes les méthodes d'un objet sont publiques, c'est-à-dire qu'on peut y accéder librement depuis l'extérieur de l'objet. Cependant il est possible de créer un attribut ou une méthode privé(e). Il suffit pour cela de lui donner un nom qui commence par deux caractères de soulignement (« underscores »), par exemple `__a` ou `__x_max`.

```
1  class Zbong  :
2      m = 45
3      __p=23
4
5  x = Zbong()
6  print(x.m)
7  print(x.__p)
```

À l'exécution, on obtient bien l'affichage de la valeur de l'attribut `m` de l'objet `x`, par contre, la ligne 7 provoque l'affichage du message « Zbong' object has no attribute '__p' ». L'attribut `__p` est privé, il ne peut être utilisé qu'à l'intérieur de l'objet `x`.

2.3.4 Les constructeurs

L'instanciation d'une classe, c'est-à-dire la création d'un objet d'une classe donnée, n'est autre que l'appel d'une fonction particulière, appelée constructeur. Par défaut, en l'absence d'instructions spécifiques dans la programmation, Python ajoute automatiquement un constructeur minimal lors de la génération d'une classe. Celui-ci n'apparaît pas explicitement.
Il est nécessaire d'étendre ce constructeur minimal pour obtenir des classes et des objets plus intéressants. Pour cela on réécrit le constructeur qui s'appellera toujours `__init__(self, var1, var2, ..., varN)`, où `var1`, `var2`, etc. , sont les variables

dont on aura besoin. Supposons, pour l'exemple, que l'on veuille créer des objets « rectangles » de longueur et de largeur variables mais tous de la même couleur rouge.

```
1  class Rectangle  :
2      couleur = 'red'
3      def __init__(self, L, l)  :
4          self.longueur = L
5          self.largeur = l
6
7  rectA = Rectangle(14,5)
8
9  print(rectA)
10 print(rectA.longueur)
11 print(rectA.largeur)
12 print(rectA.couleur)
13
14 rectB = Rectangle(100,9)
15
16 print(rectB)
17 print(rectB.longueur)
18 print(rectB.largeur)
19 print(rectB.couleur)
```

La ligne 2 crée l'attribut `couleur`. Celui-ci aura la valeur 'red' pour tous les objets de la classe `Rectangle` au moment de leur création.
Les lignes 3 à 5 décrivent le constructeur : lorsqu'on instanciera un objet de la classe `Rectangle` il faudra donner sa longueur et sa largeur.
À la ligne 7 on crée un objet de la classe `Rectangle` de longueur 14 et de largeur 5.
L'exécution des lignes 9 à 12 provoque l'affichage suivant :

```
  <__main__.Rectangle object at 0x7f45ee0247b8>
4
5
red
```

La première ligne de cet affichage dit que `RectA` est un objet `Rectangle` et qu'il est accessible dans la mémoire à l'adresse 0x7f45ee0247b8. Les lignes suivantes affichent les valeurs des trois attributs de l'objet `RectA`.

À la ligne 14 on crée un deuxième objet `Rectangle`. Les dimensions de celui-ci sont différentes alors que la couleur reste la même.

Exercice 1.1

Quels sont les affichages provoqués par l'exécution du programme suivant ?

```
class Rectangle  :
    couleur = 'red'
    def __init__(self, L, l)  :
        self.longueur = L
        self.largeur = l

class Disque  :
    couleur = 'orange'
    def __init__(self, r)  :
        self.rayon = r

class LettreI  :
    def __init__(self, l)  :
        self.baton = Rectangle(2*l,4*l)
        self.baton.couleur = 'blue'
        self.point = Disque(l)
        self.point.couleur = 'blue'

lettrine = LettreI(0.7)

print(lettrine)
print(lettrine.baton)
print(lettrine.baton.couleur)
print(lettrine.baton.longueur)
print(lettrine.baton.largeur)
print(lettrine.point)
print(lettrine.point.couleur)
print(lettrine.point.rayon)
```

Exercice 1.2

Rédigez le code Python d'une classe `Case` avec :

- deux attributs appelés `abscisse` et `ordonnée`,
- un constructeur qui donne aux attributs les valeurs passées en arguments, ou, par défaut, 0,
- une méthode qui retourne la valeur de l'`abscisse`,
- une méthode qui affiche la valeur des deux attributs.
- une méthode qui remplace la valeur de l'attribut `abscisse` par le nombre donné en argument,