Chapitre 3

Modularité

1. Importations dans Python

1.1. Présentation et définitions

En général, dans l'écriture d'un programme, il n'est pas question de réécrire ce qui existe déjà : ce serait beaucoup trop long et improductif. C'est pour cela que de nombreux programmes commencent par importer des fonctionnalités supplémentaires. Un des intérêts du langage Python est justement la richesse de ses nombreuses bibliothèques et de ses modules dans des domaines très variés.

Définition : *un module est un fichier contenant des variables et des fonctions.*

Définition : *un package est une collection de modules sous un espace de nom commun. Il contient un fichier* __init__.py *pour indiquer à Python qu'il ne s'agit pas d'un simple répertoire, mais d'un package.*

Définition : *une bibliothèque est constituée d'un ou plusieurs* packages *(ensemble de dossiers et de sous-dossiers) contenant plusieurs modules.*

Remarque : bibliothèque se traduit en anglais par *library*, à ne pas confondre avec une librairie qui se traduit en anglais par *bookshop* ou *bookstore*.

Une bibliothèque standard est distribuée avec Python. Il est possible d'importer des bibliothèques supplémentaires selon les besoins.

Définition : *une API (*Application Programming Interface*) est une interface permettant à des programmes d'interagir avec une bibliothèque sans en avoir un accès direct.*

C'est la spécification d'un protocole, en quelque sorte, qui permet de communiquer avec la bibliothèque.

1.2. Syntaxe des importations

La syntaxe minimale pour importer un module utilise la fonction import.

Exemple : import de toutes les fonctions de la bibliothèque *numpy*.

```
1 import numpy
```

Remarque : avec cette syntaxe, pour utiliser une des fonctions du module, il faut faire précéder le nom de la fonction par le nom du module.

Exemple : erreur car la fonction *cos* n'est pas définie par défaut dans Python.

```
1 import numpy
2 print(cos(0))
```

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 import numpy
2 print(numpy.cos(0))
```

Pour éviter de devoir saisir le nom complet du module, il est possible d'utiliser un alias avec la synthaxe import ... as ....

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 import numpy as np
2 print(np.cos(0))
```

La synthaxe from ... import ... *permet de n'importer qu'une ou plusieurs fonctions et évite de devoir saisir le nom du module.*

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 from numpy import cos
2 print(cos(0))
```

La syntaxe précédente peut permettre d'importer toutes les fonctions tout en évitant de devoir saisir le nom du module : from ... import $*$. *Cette possibilité est cependant vivement déconseillée.*

Exemple : le code suivant fonctionne et affiche bien *1.0*.

```
1 from numpy import *
2 print(cos(0))
```

1.3. Bibliothèques classiques

Pour un usage pédagogique, nous avons utilisé la bibliothèque *turtle*. Il permet de découvrir de façon ludique la programmation, dont la programmation orientée objet et la programmation événementielle.

Les bibliothèques *numpy* et *scipy* sont utilisées pour le calcul scientifique, *matplotlib* pour tout ce qui concerne les graphiques, *pil* pour la manipulation et le traitement d'images, *tkinter* pour obtenir des outils pour la création d'interfaces graphiques, *pygame* pour la création de jeux en 2D, etc.

Remarque : dans le cas où l'utilisateur créer un module qu'il souhaite ensuite importer, il faut préciser à l'interpréteur Python le chemin d'accès au fichier. La syntaxe sera du type :

```
1 from os import chdir
2 chdir(''C:/users/Toto/docs/programmes'')
```

Une alternative, plus simple, est de placer dans le même dossier le fichier du module et le fichier du programme réalisant l'importation.

2. Modules, interfaces et encapsulation

2.1. Découpage et réutilisation de code

Une des clefs du développement à grande échelle consiste à circonscrire et séparer proprement les différentes parties du programme. Typiquement, l'interface graphique

est séparée du cœur de l'application. *Les grands programmes doivent être fractionnés en plusieurs parties (briques), c'est le principe de la modularité.*

Chaque partie (brique) est un module. Les fonctionnalités définies dans un des modules peuvent être utilisées dans un autre module. On a donc tout intérêt à ce que chaque module soit dédié à la résolution d'une tâche, ou d'un ensemble de tâches apparentées, clairement identifiable.

Les avantages sont que le programme est :

- lisible ;
- portable ;
- réutilisable (une brique peut servir à plusieurs programmes) ;
- facile à maintenir et à modifier.

2.2. Interfaces

Un module possède une partie publique appelée interface et une partie privée appelée implémentation (ou réalisation).

L'interface énumère les fonctions définies dans le module et qui sont destinées à être utilisées dans la réalisation d'autres modules, appelés clients. *L'interface est donc liée à sa documentation : pour chaque fonction il faut connaître son rôle, son nom, la liste de ses paramètres et sa spécification*, c'est à dire les conditions auxquelles la fonction peut être appliquée et les résultats à attendre. Éventuellement, des informations concernant le temps d'exécution ou l'espace mémoire peuvent être utiles.

L'objectif de l'interface est de permettre à un utilisateur de faire appel aux fonctionnalités du module sans avoir besoin de consulter son code.

L'implémentation peut être plus ou moins complexe : l'auteur du module est libre de coder à sa manière. Il doit juste respecter le cahier des charges des fonctions décrites dans l'interface.

Remarque : en Python, une variable *__name__* est crée automatiquement à l'exécution d'un programme. Elle contient le nom du programme courant quand on importe un programme, mais le nom du programme principal est toujours *__main__*. Par conséquent, les instructions placées après la condition suivante ne sont exécutées que si le module est le programme principal :

```
1 if __name__ = "__main__":
2     bloc d'instructions
```

C'est utile, par exemple, pour placer des tests importants pour l'implémentation du module, mais qui ne font pas partie de son interface.

2.3. Encapsulation

L'auteur d'un module, pour créer son implémentation, est libre d'utiliser les structures de données qu'il souhaite, de définir des fonctions, des objets, etc., qui ne sont pas inclus dans l'interface. Ces éléments internes, souvent qualifiés de « détails d'implémentations », peuvent constituer une large part du code du module.

Tous ces éléments, hors de l'interface, sont qualifiés de privés et ne doivent pas être utilisés par les modules clients. On parle à leur propos d'encapsulation, pour signifier qu'ils sont comme enfermés dans une boîte hermétique que l'utilisateur ne doit pas ouvrir.

L'intérêt de l'encapsulation est de diminuer le couplage entre les modules. Ainsi, l'auteur d'un module peut faire évoluer son code, sans que les codes des modules clients n'aient besoin d'être adaptés.

Remarque : de nombreux langages de programmation, adaptés aux projets à grande échelle, permettent un contrôle strict de l'encapsulation, rendant l'accès aux éléments privés très compliqué voire impossible.

En Python, une telle fonctionnalité n'existe pas. En revanche, l'auteur d'un module peut indiquer que certains éléments (variables globales et fonctions) sont privés en faisant commencer le nom par « _ ». Par convention, tous les autres éléments sont publics et doivent être compris comme appartenant à l'interface.

Capacités exigibles

- Utiliser des API (*Application Programming Interface*) ou des bibliothèques.
- Exploiter leur documentation.
- Créer des modules simples et les documenter.

Chapitre 4

Structures de données linéaires

1. Notion de structures de données

1.1. Généralités

Rappel : en informatique, une variable a un nom et correspond à de l'information placée en mémoire. Le type de données (ou simplement type) est contenu dans cette information : une valeur appartenant à un ensemble précis (nombre entier ou flottant, booléen, etc.) et l'ensemble des opérations autorisées avec cette valeur.

En Python, des types de données simples (*int*, *float*, *bool*, *str*) ou construits (*list*, *tuple*, *dict*, etc.) sont implémentés. Il n'y a pas besoin de savoir comment ces types sont représentés dans la machine pour pouvoir les utiliser.

Afin d'organiser et traiter des données, les manipuler avec des algorithmes, et donc obtenir un traitement informatique efficace, on peut définir une structure de données. Celle-ci peut regrouper des objets de type différents et permet de stocker les données d'une manière particulière.

Définition : *un type abstrait de données (ou une structure de données abstraite) est un ensemble de données avec l'ensemble des opérations permises.*

C'est en quelque sorte un cahier des charges : on annonce quelles sont les données et on précise comment on les manipule, quelles sont les actions autorisées. *Ceci ne dépend pas du langage de programmation qui est utilisé.*

La réalisation concrète, l'implémentation, dépend en revanche du langage de programmation. *Quelle que soit l'implémentation choisie, les fonctions de l'interface doivent donner les mêmes résultats.* Cependant, la connaissance de l'implémentation en machine des différentes opérations et de la représentation de l'objet a son importance. Elle permet d'évaluer les coûts, en temps et en espace, des algorithmes que nous programmons.

Exemples : Python permet d'utiliser directement des chaînes de caractères, car le type *str* est un type de base. En revanche, dans d'autres langages il faut construire des tableaux de caractères, car les types de base sont les caractères et les tableaux.

Au sein d'un langage, plusieurs implémentations peuvent être possibles. Pour des structures complexes, en Python, il est notamment possible d'utiliser les *listes* ou définir des classes.

Une structure de donnée est construite à partir de structures élémentaires. Lorsque c'est fait, cette structure peut servir à la construction de nouvelles structures, de plus en plus complexes.

1.2. Caractéristiques

Définition : *une structure est dite linéaire lorsque les éléments sont ordonnés, s'il y a une chronologie.*

Sinon, il n'y a pas d'ordre, mais des relations entre les éléments, la structure est dite non linéaire.

Définition : *lorsque les éléments sont tous du même type, on parle de structure homogène.*

Définition : *lorsque la quantité de mémoire utilisée est fixe, on parle de structure statique.*

Lorsque la quantité de mémoire est variable, on parle de structure dynamique.

Exemples : structures rencontrées en première et en terminale.

Structures élémentaires	Structures complexes	
Entier	Tableau	Linéaire
Flottant	Liste chaînée	Linéaire
Booléen	Pile	Linéaire
Caractère	File	Linéaire
	Arbre	Non linéaire
	Graphe	Non linéaire

Un tableau est une structure indexée, alors qu'une liste chaînée, un arbre et un graphe sont des structures récursives.

Le choix de la structure de données doit se faire en fonction du type des informations utilisées, de la manière de stocker une donnée, de la quantité de mémoire à utiliser. *La structure doit être adaptée à des algorithmes qui utilisent des opérations comme la lecture, l'insertion la suppression de donnée.*

1.3. Opérations du TAD

L'interface est constituée par l'ensemble des opérations que possède le TAD. On peut classer ces opérations en trois types.

- *Les constructeurs.* Ils permettent de créer le TAD.
- *Les transformateurs.* Ils permettent de modifier l'objet : ajouter, effacer, accéder aux données, modifier une donnée, etc.
- *Les observateurs.* Ils donnent des informations sur l'état de l'objet : rechercher une donnée, connaître le nombre d'éléments, etc.

Les opérations élémentaires qui sont toujours présentes sont :

- *ajouter* une donnée à l'objet ;
- *lire* une donnée ;
- *modifier* une donnée ;
- *supprimer* une donnée.

Remarque : on utilise parfois l'acronyme *CRUD* (*Create*, *Read*, *Update*, *Delete*), pour créer, lire, mettre à jour et supprimer.

On peut également ajouter *rechercher* une donnée.

2. Structures linéaires : tableaux et listes chaînées

2.1. Présentation

Une structure linéaire sur un ensemble E est une suite d'éléments de E, où chaque élément a une place bien précise.

Soit les éléments ont chacun un indice qui permet d'accéder directement à cette place, soit chaque place a un successeur ou un prédécesseur, et l'accès à cette place n'est plus direct.

- *Dans le premier cas, on parle de tableau. L'avantage est la rapidité d'accès à un élément particulier.*
- *Dans le second cas, on parle de liste chaînée. L'avantage est le dynamisme de la structure.*

Remarque : une liste en Python, de type *list* est une structure hybride, présentant des caractéristiques d'un tableau et d'une liste chaînée, car elle est indexée et dynamique.

2.2. Le tableau

Si tous les éléments du tableau sont du même type, ils occupent tous la même taille t en mémoire. Il suffit alors de stocker l'adresse en mémoire a du premier élément pour pouvoir accéder à n'importe quel élément d'indice k, en effet son adresse en mémoire sera $a + k \cdot t$. *Tous les éléments sont donc accessibles avec un coût constant, le temps de calcul de l'adresse et l'accès à cette adresse.*

La place du tableau en mémoire est réservée à la création : $n \cdot t$ pour un nombre n d'éléments. Une contrainte est l'impossibilité de remplacer un élément par un élément d'un autre type ou d'agrandir la taille du tableau, car on ne sait pas ce qu'il y a en mémoire après la case contenant le dernier élément.

Remarque : ces caractéristiques correspondent à des tableaux statiques.

2.3. La liste chaînée

Un élément d'une liste chaînée est stockée en mémoire avec l'adresse de l'élément suivant. Donc, pour accéder au n^{e} élément, il faut passer par les $n - 1$ éléments qui le précèdent. *Le coût d'accès à un élément est alors linéaire en fonction de la position de l'élément.*

En revanche, on peut ajouter des éléments au début, à l'intérieur ou à la fin de la liste, dont la taille peut varier. Pour ajouter un élément au début de la liste, pratiquement en temps constant, on place dans une case mémoire une donnée avec l'adresse du premier élément de la liste, qui devient l'élément suivant de la donnée, donc le deuxième de la liste.

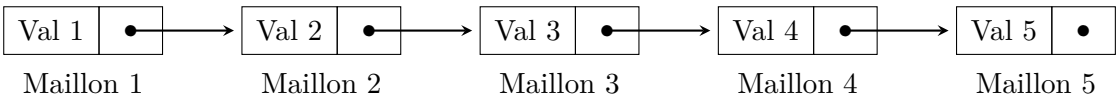

Figure 4.1 – Liste chaînée

3. Structures linéaires : piles et files

3.1. La pile

La pile (« stack » en anglais) est un type abstrait de données dynamique dans lequel toute insertion ou suppression d'un élément se fait à une extrémité, appelée dessus ou sommet de la pile.

Le dernier élément entré sera le premier à sortir : on parle de LIFO (« Last In First Out » en anglais).

On prend souvent l'analogie avec une pile d'assiettes sur une étagère : la seule assiette directement accessible est la dernière assiette qui a été déposée sur la pile.

Exemples :

- pile des appels en récursivité ;
- la vérification syntaxique (nombres de parenthèses ouvrantes et fermantes dans une expression...) s'applique facilement avec le principe d'une pile ;
- très souvent, de nombreux logiciels, navigateurs, etc. possèdent un bouton « retour en arrière » qui permet de revenir au précédent état du document avant la dernière modification, ces fonctionnalités sont traitées à l'aide du principe des piles ;
- etc.

Propriété : la pile permet d'ajouter des éléments sans se préoccuper du nombre d'éléments déjà présents, et sans savoir a priori combien d'éléments on va finalement ajouter. En principe, la pile n'a pas de taille maximale.

En pratique, il peut y avoir un débordement lorsque la capacité maximale est dépassée.

Propriété : opérations de base sur les piles.

- Une fonction qui crée une pile vide.
- Une fonction qui ajoute un élément sur la pile : *empiler*, en anglais *push*.
- Une fonction qui retire l'élément sur la pile, le sommet, et le renvoie : *dépiler*, en anglais *pop*.

D'autres fonctions peuvent fournir des informations sur la pile. Les plus utilisées sont :

- une fonction qui permet de savoir si la pile vide ;
- une fonction qui donne la taille de la pile vide ;
- une fonction qui donne le sommet de la pile (sans supprimer l'élément).

3.2. La file

La file (« queue » en anglais) est un type abstrait de données dynamique dans lequel l'insertion d'un élément se fait à une extrémité et la suppression se fait à l'autre extrémité.

Le premier élément entré sera le premier à sortir : on parle de FIFO (« First In First Out » en anglais).

On prend souvent l'analogie avec une file d'attente à un guichet par exemple : le premier arrivé sera le premier à sortir de la file d'attente et à atteindre le guichet.

Exemples : gestion de processus par un système d'exploitation. Dans un ordinateur, lorsque des appels aux processus « 1 », « 2 », puis « 3 » se succèdent, l'ensemble est

stocké dans une table des processus, propre au noyau du système d'exploitation. Lorsque le « processus 1 » se termine, le système sait qu'il doit passer au « processus 2 », puis au « processus 3 ».

Un autre exemple est le cas d'une imprimante qui gère une file de documents en attente d'impression.

Propriété : la file est une structure dynamique permet d'ajouter des éléments sans se préoccuper du nombre d'éléments déjà présents, et sans savoir a priori combien d'éléments on va finalement ajouter.

Propriété : opérations de base sur les files.

- Une fonction qui crée une file vide.
- Une fonction qui ajoute un élément en queue de la file : *enfiler*, en anglais *push*.
- Une fonction qui retire l'élément en tête de la file et le renvoie : *défiler*, en anglais *pop*.

D'autres fonctions peuvent fournir des informations sur la pile. Les plus utilisées sont :

- une fonction qui permet de savoir si la file vide ;
- une fonction qui donne la taille de la file vide ;
- une fonction qui donne la tête de la file (sans supprimer l'élément).

4. Structure linéaire : le dictionnaire

Rappel : *revoir le cours de première sur les dictionnaires.*

Le dictionnaire est une structure de données qui répond à un problème très fréquemment rencontré en informatique : la recherche d'information dans un ensemble géré de façon dynamique (c'est-à-dire dont le contenu est susceptible d'évoluer au cours du temps).

Un dictionnaire est un mappage. C'est un TAD qui permet de stocker des *valeurs* et d'y accéder au moyen d'une *clef*, contrairement au tableau qui permet d'accéder à une donnée au moyen d'un indice.

Un dictionnaire contient une collection de clefs et une collection de valeurs. Chaque clef est associée à une valeur unique. L'association entre une clé et une valeur est appelée une *paire clé-valeur* ou parfois un *item*.

Un dictionnaire représente donc un mappage de clefs vers des valeurs.

Propriété : opérations de ce TAD.

- *Ajout :* on associe une nouvelle valeur à une nouvelle clef.
- *Lecture :* on lit la valeur associée à une clef.
- *Modification :* on modifie la valeur d'un couple clef-valeur (la clef restant identique).
- *Suppression :* on supprime une clef (et donc la valeur qui lui est associée).
- *Recherche :* on recherche une valeur à l'aide de la clef associée à cette valeur.

Remarque : contrairement aux listes chaînes, aux piles et aux files, les dictionnaires sont implémentés nativement en Python.

Capacités exigibles

- Spécifier une structure de données par son interface.
- Distinguer interface et implémentation.
- Écrire plusieurs implémentations d'une même structure de données.
- Distinguer des structures par le jeu des méthodes qui les caractérisent.
- Choisir une structure de données adaptée à la situation à modéliser.
- Distinguer la recherche d'une valeur dans une liste et dans un dictionnaire.