DIU Enseigner l'informatique au lycée

# Modularité

**Exercice 1** Écrire l'interface d'un module spécialisé pour des ensembles contenant des entiers naturels (en précisant les exceptions) et donner une réalisation. □

**Exercice 2** En utilisant le rattrapage d'exception et une instruction `break`, écrire un programme qui réalise une division `x/y`, pour deux entiers saisis au clavier, et qui redemande à saisir y tant que la valeur saisie est 0. □

**Exercice 3** L'interface des tableaux de Python fournit de nombreuses opérations à l'allure anodine mais cachant une complexité non négligeable. Rien de tel que d'essayer de réaliser soi-même ces fonctions pour s'en rendre compte. Écrire un module réalisant l'interface suivante

| fonction | description |
|---|---|
| `tranche`($t$, $i$, $j$) | renvoie un nouveau tableau contenant les éléments de $t$ de l'indice $i$ inclus à l'indice $j$ exclu (et le tableau vide si $j \leq i$) |
| `concatenation`($t_1$, $t_2$) | renvoie un nouveau tableau contenant, dans l'ordre, les éléments de $t_1$ puis les éléments de $t_2$ |

sans utiliser les opérations `+` et $t$`[`$i$`:`$j$`]` du langage. Notez qu'aucune de ces fonctions ne doit modifier les tableaux passés en paramètres. □

**Exercice 4** Voici une interface minimale pour une structure de dictionnaire.

| fonction | description |
|---|---|
| `cree()` | crée et renvoie un dictionnaire vide |
| `cle`($d$, $k$) | renvoie `True` si et seulement si le dictionnaire $d$ contient la clé $k$ |
| `lit`($d$, $k$) | renvoie la valeur associée à la clé $k$ dans le dictionnaire $d$, et `None` si la clé $k$ n'apparaît pas |
| `ecrit`($d$, $k$, $v$) | ajoute au dictionnaire $d$ l'association entre la clé $k$ et la valeur $v$, en remplaçant une éventuelle association déjà présente pour $k$ |

On veut réaliser cette interface de dictionnaire avec un tableau de couples clé-valeur, en faisant en sorte qu'aucune clé n'apparaisse deux fois.

1. Écrire un module réalisant cela.
2. La description de l'une des quatre fonctions de notre interface ne correspond pas exactement à l'opération équivalente des dictionnaires de Python. Laquelle ? Quelle expérience faire pour le confirmer ? Corriger la description pour se rapprocher de celle de Python et adapter la réalisation.

□

## Programmation objet

**Exercice 5** Écrire une classe `Ensemble` pour manipuler des ensembles (cf. exercices sur les modules) et récrire la fonction `contient_doublon` en utilisant cette classe. □

**Exercice 6** Définir une classe `Fraction` pour représenter un nombre rationnel. Cette classe possède deux attributs, `num` et `denom`, qui sont des entiers et désignent respectivement le numérateur et le dénominateur. On demande que le dénominateur soit plus particulièrement un entier strictement positif.

- Écrire le constructeur de cette classe. Le constructeur doit lever une `ValueError` si le dénominateur fourni n'est pas strictement positif.
- Ajouter une méthode `__str__` qui renvoie une chaîne de caractères de la forme `"12 / 35"`, ou simplement de la forme `"12"` lorsque le dénominateur vaut un.
- Ajouter des méthodes `__eq__` et `__lt__` qui reçoivent une deuxième fraction en argument et renvoient `True` si la première fraction représente respectivement un nombre égal ou un nombre strictement inférieur à la deuxième fraction.
- Ajouter des méthodes `__add__` et `__mul__` qui reçoivent une deuxième fraction en argument et renvoient une nouvelle fraction représentant respectivement la somme et le produit des deux fractions.
- Tester ces opérations.

Bonus : s'assurer que les fractions sont toujours sous forme réduite. □

**Exercice 7** Définir une classe `Intervalle` représentant des intervalles de nombres. Cette classe possède deux attributs `a` et `b` représentant respectivement l'extrémité inférieure et l'extrémité supérieure de l'intervalle. Les deux extrémités sont considérées comme incluses dans l'intervalle. Tout intervalle avec `b < a` représente l'intervalle vide.

- Écrire le constructeur de la classe `Intervalle` et une méthode `est_vide` renvoyant `True` si l'objet représente l'intervalle vide et `False` sinon.
- Ajouter des méthodes `__len__` renvoyant la longueur de l'intervalle (l'intervalle vide a une longueur `0`) et `__contains__` testant l'appartenance d'un élément `x` à l'intervalle.
- Ajouter une méthode `__eq__` permettant de tester l'égalité de deux intervalles avec `==` et une méthode `__le__` permettant de tester l'inclusion d'un intervalle dans un autre avec `<=`. Attention : toutes les représentations de l'intervalle vide doivent être considérées égales, et incluses dans tout intervalle.

- Ajouter des méthodes `intersection` et `union` calculant respectivement l'intersection de deux intervalles et le plus petit intervalle contenant l'union de deux intervalles (l'intersection est bien toujours un intervalle, alors que l'union ne l'est pas forcément). Ces deux fonctions doivent renvoyer un nouvel intervalle sans modifier leurs paramètres.
- Tester ces méthodes.

□

## Programmation récursive

**Exercice 8** Donner une définition récursive qui correspond au calcul de la fonction factorielle $n!$ définie par $n! = 1 \times 2 \times \cdots \times n$ si $n > 0$ et $0! = 1$, puis le code d'une fonction `fact(n)` qui implémente cette définition. □

**Exercice 9** Soit $u_n$ la suite d'entiers définie par

$$\begin{array}{rcll} u_{n+1} & = & u_n/2 & \text{si } u_n \text{ est pair,} \\ & = & 3 \times u_n + 1 & \text{sinon.} \end{array}$$

avec $u_0$ un entier quelconque plus grand que 1.

Écrire une fonction récursive `syracuse(u_n)` qui affiche les valeurs successives de la suite $u_n$ tant que $u_n$ est plus grand que 1.

La conjecture de Syracuse affirme que, quelle que soit la valeur de $u_0$, il existe un indice $n$ dans la suite tel que $u_n = 1$. Cette conjecture défie toujours les mathématiciens. □

**Exercice 10** On considère la suite $u_n$ définie par la relation de récurrence suivante, où $a$ et $b$ sont des réels quelconques :

$$u_n = \begin{cases} a \in \mathbb{R} & \text{si } n = 0 \\ b \in \mathbb{R} & \text{si } n = 1 \\ 3u_{n-1} + 2u_{n-2} + 5 & \forall n \geq 2 \end{cases}$$

Écrire une fonction récursive `serie(n, a, b)` qui renvoie le `n`-ème terme de cette suite pour des valeurs `a` et `b` données en paramètres. □

**Exercice 11** Écrire une fonction récursive `boucle(i,k)` qui affiche les entiers entre `i` et `k`. Par exemple, `boucle(0,3)` doit afficher `0 1 2 3`. □

**Exercice 12** Écrire une fonction récursive `pgcd(a, b)` qui renvoie le PGCD de deux entiers `a` et `b`. □

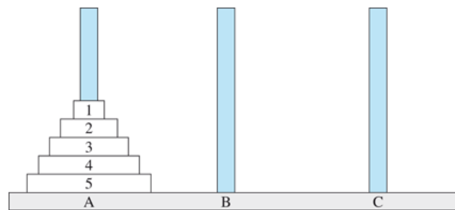
**Exercice 13** Écrire une fonction récursive `nombre_de_chiffres(n)` qui prend un entier positif ou nul `n` en argument et renvoie son nombre de chiffres. Par exemple, `nombre_de_chiffres(34126)` doit renvoyer `5`. □

**Exercice 14** En s'inspirant de l'exercice 13, écrire une fonction récursive `nombre_de_bits_1(n)` qui prend un entier positif ou nul et renvoie le nombre de bits valant `1` dans la représentation binaire de `n`. Par exemple, `nombre_de_bits_1(255)` doit renvoyer `8`. □

**Exercice 15** Écrire une fonction récursive `appartient(v, t, i)` prenant en paramètres une valeur `v`, un tableau `t` et un entier `i` et renvoyant `True` si `v` apparaît dans `t` entre l'indice `i` (inclus) et `len(t)` (exclu), et `False` sinon. On supposera que `i` est toujours compris entre `0` et `len(t)`. □

**Exercice 16** Les tours de Hanoï sont un jeu de réflexion consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :

- On ne peut déplacer plus d'un disque à la fois.
- On ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.
- On suppose que cette dernière règle est également respectée dans la configuration de départ.




Écrire une fonction `hanoi(n)` qui résoud le problème des tours de Hanoï en affichant les mouvements à effectuer, comme ci-dessous (où on utilise les nombres 1, 2 et 3 pour désigner les tours). L'argument `n` indiquera le nombre de disques à déplacer entre la tour de départ et la tour d'arrivée.

```
>>> hanoi(3)
 Move  1  on 3
 Move  1  on 2
 Move  3  on 2
 Move  1  on 3
 Move  2  on 1
 Move  2  on 3
 Move  1  on 3
```

□

**Exercice 17** Le triangle de Pascal (nommé ainsi en l'honneur du mathématicien Blaise Pascal) est une présentation des coefficients binomiaux sous la

forme d'un triangle défini ainsi de manière récursive :

$$C(n,p) = \begin{cases} 1 & \text{si } p = 0 \text{ ou } n = p, \\ C(n-1,p-1) + C(n-1,p) & \text{sinon.} \end{cases}$$

Écrire une fonction récursive `C(n,p)` qui renvoie la valeur de $C(n,p)$, puis dessiner le triangle de Pascal à l'aide d'une double boucle **for** en faisant varier `n` entre 0 et 10. □

**Exercice 18** La courbe de Koch est une figure qui s'obtient de manière récursive. Le cas de base à l'ordre 0 de la récurrence est simplement le dessin d'un segment d'une certaine longueur $l$, comme ci-dessous (figure de gauche).

Le cas récursif d'ordre $n$ s'obtient en divisant ce segment en trois morceaux de même longueur $l/3$, puis en dessinant un triangle équilatéral dont la base est le morceau du milieu, en prenant soin de ne pas dessiner cette base. Cela forme une sorte de chapeau comme dessiné sur la figure de droite ci-dessus. On réitère ce processus à l'ordre $n-1$ pour chaque segment de ce chapeau (qui sont tous de longueur $l/3$). Par exemple, les courbes obtenues à l'ordre 2 et 3 sont données ci-dessous (à gauche et à droite, respectivement).

Écrire une fonction `koch(n, l)` qui dessine avec `Turtle` un flocon de Koch de profondeur `n` à partir d'un segment de longueur `l`. □