**Université de Caen** **Année 2006/2007**
**UFR de Sciences** **1ère session/janvier**

# Licence L2
# Programmation orientée objet
## (Introduction au Python) : **EI31T**

Durée: 2H. Documents autorisés : **AUCUN.**

**Chaque candidat doit, au début de l'épreuve, porter son nom dans le coin de la copie qu'il cachera par collage après avoir été pointé. Il devra en outre porter son numéro de place sur chacune des copies, intercalaires, ou pièces annexées.**

1. Vous connaissez tous les paires de réels, qui sont des nombres complexes... En XIX siècle Clifford a inventé un autre corps numérique, les ***nombres duals***, un peu plus simple ; un nombre dual $p$ est une paire de réels $a$ et $b$, écrit comme $p = a + b\epsilon$ avec les règles arithmétiques typiques (par exemple on ajoute séparement les composantes réelle et duale). L'unité duale $\epsilon$ est irréductible, indépendante des réels (comme l'unité imaginaire dans le monde des nombres complexes), mais elle respecte l'identité : $\epsilon^2 = 0$. Donc, $(a + b\epsilon) \cdot (x + y\epsilon) = ax + (ay + bx)\epsilon$. **Écrire en Python une classe qui représente de tels nombres, avec les méthodes standard : `__init__` et `__repr__`**, et les quatres opérations arithmétiques : **`+ - * /`**. Pour la division, vérifier que $(a+b\epsilon)\cdot(a-b\epsilon)$ est un nombre réel, et coder-là de manière similaire à des nombres complexes. La méthode **`__init__`** doit pouvoir prendre 1 ou deux arguments, dans le cas d'un seul, le coefficient de $\epsilon$ est considéré nul.

2. Écrire en Python un *générateur* **`powerset(l)`**, qui accepte une liste **`l`** comme argument, et qui retourne (par **`yield`**, bien sûr) une par une, la séquence de toutes les sous-listes de la liste **`l`**. Donc, **`powerset([1,2,3])`** doit engendrer (l'ordre n'est pas spécifié) les listes : **`[], [1], [2], [3], [1,2], [1,3], [2,3], [1,2,3]`**, au total $8 = 2^3$. L'algorithme est récursif, et il est basé sur une simple observation. Pour tout élément de la liste-argument, on *peut accepter* cet élément, et aussi *ne le pas accepter*. Donc, on construit le générateur secondaire pour la queue de la liste, et on génére deux solutions pour chacune offerte par ce générateur secondaire : sans ou avec la tête de la liste.

3. **Le code de César**. Construire une classe de « pseudo-dictionnaires », des objets qui se comportent comme des dictionnaires indexés par des chaînes de caractères, par exemple **`d['yeux']`** donne **`'bhxa'`**. Cependant, **`d`** *n'est pas un vrai dictionnaire*, et on ne peut lui pas ajouter des nouvelles entrées, et il n'accepte que des chaînes. C'est une fonction déguisée en dictionnaire... Elle prend la chaîne-argument, caractère par caractère, ajoute 3 à son code, et reconstruit le caractère résultant, et la chaîne résultante. Si le code du caractère dépasse le code de **`'z'`**, on boucle : **`'z'`** est suivi par **`'a'`**, **`'b'`**, etc. Pour simplicité, ignorons les lettres majuscules et les lettres accentuées (on considère uniquement les 26 lettres latines). Prévoir une paramétrisation globale par une variable de classe – le décalage, pas obligatoirement égal à 3, mais quelconque. Utiliser les fonctions : **`ord`** qui convertit un caractère en un entier, et **`chr`** – la conversion inverse. Vous pourrez avoir besoin de l'opérateur modulo : **`%`**. Construire conceptuellement d'abord la fonction qui effectue le transcodage, et ensuite la classe des objets qui se comportent comme des dictionnaires indexables.

***Attention :*** **Je vous prie d'écrire de manière lisible, tout texte difficile à déchiffrer risque d'être rejeté. Ne jamais copiez des morceaux de code inutile, tout code hors sujet (surtout : tout le code des notes du cours, etc.) sera *pénalisé !* Évitez toute discussion qualitative (« philosophique ») inutile. Par contre, *commentez votre code*, sinon il risque d'être peu compréhensible.**