Présentation de quelques classes et fonctions associées

Depuis le début de l'année, nous avons pu voir qu'il était très facile d'utiliser le langage Python pour construire nos programmes. En particulier, il ne nécessite pas de **déclarer** au préalable les variables qui seront utilisées : on parle de **typage dynamique**, c'est à dire que pour chaque variable, l'interpréteur identifie sa **classe** à l'assignation et ceci en fonction des commandes utilisées.

On fera donc attention aux différentes classes des objets mais aussi à leurs **attributs** (des propriétés propres à la classe de l'objet) et aux nombreuses **méthodes associées** (des fonctions particulières définies pour les objets d'une même classe).

Quelques classes pour manipuler des données scalaires

- A ce titre, la première classe est celle des **nombres entiers** : ce sont les objets du type `int` et parmi les opérations associées les plus courantes, on distinguera :

commande Python	interprétation
`x//y`	renvoie le quotient de la division euclidienne de x par y
`x%y`	renvoie le reste de la division euclidienne de x par y
`divmod(x,y)`	renvoie le couple $(quotient, reste)$ de la division euclidienne de x par y
`range(x)`	renvoie la liste des entiers $[0, 1, \dots, x-1]$

  ainsi que toutes les opérations usuelles. Pour finir, on pourra toujours convertir un nombre réel en entier par la commande `int`.

```
In : int(2.5); int(-1.6)

                                    2
                                    -1
```
python

- La seconde classe utile est celle des **nombres réels** : ce sont les objets du type `float` et parmi les opérations associées les plus courantes, on distinguera :

commande Python	interprétation
`round(x)` ou `round(x,n)`	renvoie la valeur arrondie de x à 10^{-n} près
`abs(x)`	renvoie la valeur absolue de x

  ainsi que toutes les opérations usuelles. La plupart du temps, on utilisera également les méthodes intégrées à la librairie **math**, ne serait-ce que pour faire appel aux fonctions mathématiques :

  `acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`, `cos`, `cosh`, `exp`, `log`, `log10`, `sin`, `sinh`, `sqrt`, `tan`, `tanh` et `e`, `pi`

  Mais on retiendra également l'existence de commande Python plutôt pratiques :

commande Python	interprétation
`floor(x)`	renvoie le plus petit entier inférieur ou égal à x
`ceil(x)`	renvoie le plus petit entier supérieur ou égal à x
`factorial(x)`	renvoie la valeur de $x!$ à condition que x soit un entier naturel

  Pour finir, on pourra toujours convertir un nombre réel en entier par la commande `float`.

```
In : float(2)

                                    2.0
```
python

- Une dernière classe d'objets est aussi très utile : il s'agit des **objets booléens** du type `bool` et qui ne prennent que deux valeurs `True` ou `False`. Ces valeurs nous permettent de vérifier des conditions logiques qu'elles soient simples, ou multiples. Pour cela, on veillera à bien connaître les connecteurs logiques `and`, `or` et `not` :

```
In : 0<1/1e17; 13==26/2 and 4<6<5

                                    True
                                    False
```
python

  En fait, les résultats obtenus sont donnés par des **tables logiques** :

A	B	not A	A and B	A or B
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Bien entendu, suivant le problème donné, on pourra faire appel à d'autres classes (`complex, array`...) avec leur lot de fonctions associées et pour les exploiter, il suffira à chaque fois d'importer le **module** ou la **librairie** correspondante.

Exercice 1 - Test de primalité

On rappelle qu'un entier naturel n est premier s'il ne possède que deux diviseurs positifs 1 et lui-même.

1. Dans le langage Python, construire le programme *diviseurs* qui pour un entier naturel n donné, renvoie affiche les diviseurs de n.
2. En déduire la fonction booléenne *premier* qui pour tout entier naturel n donné, renvoie `True` si n est premier et `False` sinon. On ajoutera un test conditionnel sur le paramètre n afin de renvoyer un message d'erreur si celui-ci n'est pas entier naturel.
3. Construire alors le programme *listepremier* qui, pour tout entier n donné, renvoie la liste des n premiers nombres premiers.

Quelques classes pour manipuler des données structurées

Certaines classes nous permettront de gérer des objets constitués eux-mêmes d'autres objets, on parle plutôt de **données structurées**, et on distinguera :

- **les n-uplets du type** *tuple*
  Ils représentent la classe la plus courante et leur définition est implicite puisqu'il suffira de juxtaposer les éléments à l'aide d'une virgule, avec des parenthèses ou non.
- **les ensembles du type** *set*
  Ils représentent une classe très pratique car on y retrouve toutes les opérations ensemblistes usuels. On pourra construire de tels ensembles à l'aide d'accolades ou tout simplement par la commande de conversion `set`.
- **les chaînes de caractères du type** *string*
  Elles sont construites soit en utilisant la fonction de conversion `str`, soit à l'aide de guillemets simples ' ' ou doubles " ", selon la présence d'apostrophes dans la chaîne donnée :

```
In : s,t= 'sans apostrophe!',"avec l'apostrophe"; type(s); type(t)
                                                <class 'str'>
                                                <class 'str'>
```
python

- **les listes du type** *list*
  Les **listes** permettent de représenter des séquences modifiables. C'est le type que l'on préférera manipuler tant les méthodes sont nombreuses. On pourra définir une telle liste de trois façons :
  1. en convertissant une variable de type *tuple* grâce à la commande `list`,
  2. en complétant la liste au fur et à mesure dans un programme itératif avec une boucle `for` ou `while`,
  3. en décrivant la séquence contenue dans la liste. On parle alors de **liste par compréhension** pour laquelle les éléments sont exprimés en fonction de l'indice associé :

```
In : L = [k**2 for k in range(0,11)]; L
                                [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```
python

  Et en plus, on pourra même y ajouter des **instructions conditionnelles** :

```
In : L = [k**2 for k in range(0,11) if k%2 == 0]; L
                                [0, 4, 16, 36, 64, 100]
```
python

  Contrairement aux n-uplets, on pourra cette fois-ci en modifier le contenu et on fera bien entendu attention à l'indexation car les éléments sont toujours numérotés de 0 jusqu'à la longueur de la liste obtenue par la commande `len` -1 :

```
In : L = list(range(11)); L[5] = 0; print(L); len(L)
                                [0, 1, 2, 3, 4, 0, 6, 7, 8, 9, 10]
                                11
```
python

  Si de plus, on cherche à supprimer un élément, on pourra toujours le faire au moyen de la commande `del` :

```
In : del(L[2:5]); L
                                [0, 4, 100]
```
python

En fait, la donnée de deux indices sous la forme `i:j` nous permet d'extraire les éléments d'une liste, mais on veillera à bien comprendre que l'élément d'indice i est toujours inclus, alors que l'élément d'indice j est exclus.

Pour finir, on présente ici quelques opérations déjà rencontrées dans des programmes précédents :

commande Python	interprétation
`x in L` ou `x not in L`	teste si x appartient ou non à la liste L
`L1 + [x]`	ajoute l'élément x à la liste $L1$
`L1 + L2`	renvoie la concaténation des listes $L1$ et $L2$
`L * n`	renvoie la concaténation de n-fois la liste L
`max(L)` ou `min(L)`	renvoie le maximum ou le minimum de la liste L

Cette classe possède des fonctions plus spécifiques encore, appelées aussi **attributs**, et qui permettent d'obtenir des propriétés liées à l'objet lui-même :

commande Python	interprétation
`L.index(x)`	renvoie le premier indice de l'élément qui vaut x
`L.count(x)`	renvoie le nombre d'occurrences de x dans L

De la même façon, on retrouvera cette notation pointée dans l'utilisation des **méthodes**. Ce sont des fonctions qui opèrent sur la liste donnée :

commande Python	interprétation
`L.append(x)`	ajoute l'élément x à la fin de la liste L
`L1.extend(L2)`	ajoute à la fin de $L1$ les éléments de $L2$
`L.insert(i,x)`	insère au rang i l'élément x
`L.remove(x)`	supprime la première occurrence de x dans L
`L.reverse()`	permet de retourner la liste L en inversant les éléments
`L.sort()`	permet d'ordonner la liste L

Exercice 2 - PGCD de deux entiers

On peut toujours déterminer le PGCD de deux entiers en se ramenant à sa définition, c'est à dire :

$$PGCD(a,b) = max(D_a \cap D_b)$$

1. Dans la console interactive, définir les ensembles :

   $$A = \{1,2,3,4,5,6\} \text{ et } B = \{0,2,4,6,8\}$$

   puis taper les instructions suivantes : `A.intersection(B)`, `A.union(B)`. Que remarquez-vous ?

2. Dans le langage Python, construire la fonction *diviseurs* qui, pour tout entier n non nul donné, renvoie la liste des diviseurs de n.

3. En déduire alors le programme *pgcd* qui, pour tout couple (a,b) d'entiers non nuls, renvoie le PGCD des entiers a et b.

Exercice 3 - Recherche du maximum ou du minimum dans une liste donnée

Dans le programme précédent, on a naturellement extrait le plus grand élément d'un ensemble à l'aide de la commande `max`. On cherche ici à construire les fonctions *maximum* et *mininum* qui pour une liste L donnée, renvoie la valeur maximale et minimale de la liste L.

Par exemple, pour construire la fonction *maximum*, on initialise une constante $M = L[0]$, puis on parcourt la liste pour k allant de 1 à n afin de comparer M avec $L[k]$ et :

- si $L[k] > M$, on écrase la valeur de M
- sinon, on ne fait rien

1. Dans le langage Python, construire la fonction *maximum* qui pour une liste donnée, renvoie sa valeur maximale.
2. Modifier le programme précédent afin que celui-ci renvoie un couple (k_0, M) avec k_0 le plus petit rang contenant M, la valeur maximale.
3. Dans le langage Python, construire la fonction *minimum* qui pour une liste donnée, renvoie sa valeur maximale.
4. Modifier le programme précédent afin que celui-ci renvoie un couple (k_0, m) avec k_0 le plus petit rang contenant m, la valeur minimale.