Importation de librairies additionnelles

Nous avons qu'il était assez simple de construire nos premiers programmes dans le langage Python. Si on veut aller plus loin, il sera parfois nécessaire d'importer des **librairies additionnelles** : il s'agit généralement de modules externes qui contiennent des fonctions déjà programmées.
Par exemple, nous prendrons l'habitude d'importer la librairie **math** ajoutant les fonctions mathématiques usuelles qui ne sont pas intégrées par défaut :

```
In : import math
```
python

On peut alors obtenir une rapide description des fonctions données :

```
In : help(math)

Help on module math:
(...)
FUNCTIONS
acos(...)
acos(x)
Return the arc cosine (measured in radians) of x.
(...)
```
python

On retiendra en particulier le nom des fonctions usuelles ainsi que la donnée des constantes e et π :

`acos`, `acosh`, `asin`, `asinh`, `atan`, `atanh`, `ceil`, `cos`, `cosh`, `degrees`, `exp`, `fabs`, `factorial`, `floor`, `fsum`, `log`, `log10`, `sin`, `sinh`, `radians`, `sqrt`, `tan`, `tanh` et `e`, `pi`

Par contre, ces fonctions sont associées à la librairie importée et il conviendra de rappeler le nom du module lors de nos instructions :

```
In : math.sqrt(169)
                                                    13.0
```
python

Bien entendu, pour simplifier nos scripts, on peut éviter d'ajouter un tel préfixe et on préfèrera souvent importer le module d'une autre façon :

```
In : from math import *; sqrt(169)
                                                    13.0
```
python

Au fil de l'année, on sera donc amené à travailler avec différents modules en fonction de nos besoins et on pourra d'ores et déjà retenir l'existence des librairies suivantes :

- le module **cmath** pour travailler sur les nombres complexes,
- le module **random** permettant de générer des nombres pseudos-aléatoires,
- le module **time** pour gérer l'horloge interne du système,
- les modules **pylab** ou **matplolib** pour représenter les fonctions usuelles ou des objets géométriques,
- le module **numpy** pour travailler sur les tableaux,
- le module **scipy** apportant de nombreuses méthodes d'approximation pour la résolution de problèmes mathématiques.

Première application : tracer la courbe représentative d'une fonction donnée

Si on souhaite représenter graphiquement les fonctions usuelles, on peut donc faire appel à la librairie **pylab** qui regroupe les fonctions les plus courantes des modules **math**, **matplotlib** et **numpy**.

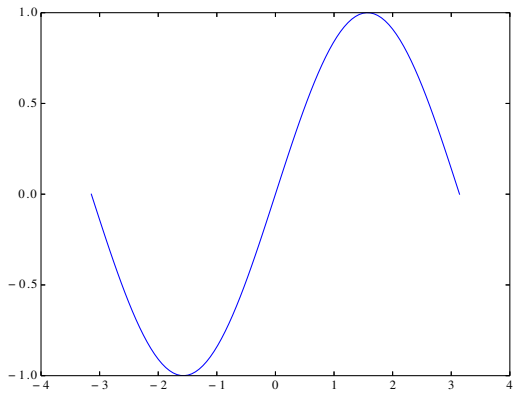
Concrètement, le logiciel représentera des points donnés par des tableaux de valeurs. Par exemple, pour représenter la fonction sin, on fera appel aux commandes `plot` et `linspace` qui détermine un nombre de points donnés dans un intervalle $[a, b]$:

```
from pylab import *

def representation(n):
    X=linspace(-pi,pi,n)
    Y=[sin(x) for x in X]
    plot(X,Y)
    show()
```
python

et ainsi, on obtient par un simple appel la courbe souhaitée sur $[-\pi, \pi]$:

```
In : representation(100)
```



Remarque La commande `linspace` est très pratique, mais on aurait aussi pu construire le tableau des abscisses **de façon itérative**. Par exemple, pour représenter la fonction carrée sur $[a, b]$, on peut écrire le script suivant :

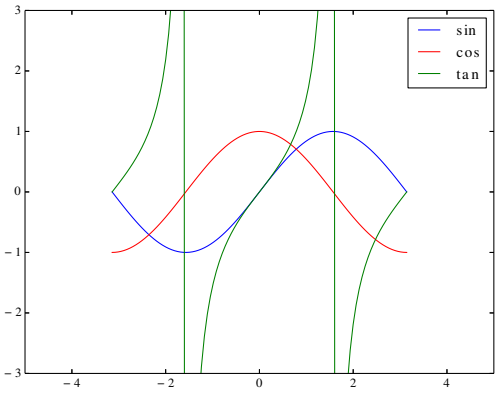
```
def carree(a,b,n):
    X=[]
    h=(b-a)/n
    for k in range(0,n+1):
        X=X+[a+k*h]
    Y=[x**2 for x in X]
    plot(X,Y)
    show()
```

Pour finir, les options sont multiples, que ce soit dans la façon de représenter les points donnés, dans le choix des couleurs, l'affichage d'un titre ou d'une légende et on n'hésitera pas à regarder l'aide à propos de la commande `plot`. Ainsi, on peut représenter toutes les fonctions trigonométriques dans une même fenêtre :

```
def trigo(n):
    X=linspace(-pi,pi,n)
    U=[sin(x) for x in X]
    V=[cos(x) for x in X]
    W=[tan(x) for x in X]
    plot(X,U,'b',label='sin')
    plot(X,V,'r',label='cos')
    plot(X,W,'g',label='tan')
    axis([-5,5,-3,3])
    legend()
    show()
```

On veillera à retenir l'utilisation de la commande `axis` qui permet d'ajuster la fenêtre d'affichage, ou alors les commandes `xlim` et `ylim` qui limitent les valeurs obtenues sur chacun des axes.

```
In : trigo(100)
```



Exercice 1 - Coefficient binomial

Soit $n \in \mathbb{N}$. On définit le coefficient binomial $\binom{n}{k}$ pour tout $k \in [\![0, n]\!]$ par :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

1. Dans le langage Python, construire la fonction *binomial* qui pour tout couple (k, n) donné, renvoie la valeur du coefficient $\binom{n}{k}$.
2. En déduire le programme $listecoefs$ qui pour tout entier n donné, renvoie la liste des coefficients binomiaux associés :

$$\{\binom{n}{k},\ k \in [\![0, n]\!]\}$$

Exercice 2 - Représentation d'une parabole

On considère la fonction f définie sur $\mathbb{R}$ par $f(x) = ax^2 + bx + c$ avec $(a, b, c) \in \mathbb{R}^3, a \neq 0$. On note C_f la parabole associée.

1. Dans le langage Python, construire la fonction *sommet* qui pour tout triplet (a, b, c) renvoie les coordonnées de S, le sommet de la parabole.
2. Construire la fonction *parabole* qui pour tout $(a, b, c, h) \in \mathbb{R}^4$ renvoie la courbe représentative de la fonction f sur l'intervalle $[x_S - h, x_S + h]$.
3. En utilisant une structure conditionnelle, modifier le programme précédent de sorte que :
   - si a est nul, le programme affiche le texte "attention, ce n'est pas un polynôme du second degré",
   - sinon, le programme renvoie le graphe associé.