Fonctions, objets, modules

I. Compléments sur les fonctions

I.1. Rappels et syntaxe alternative

On rappelle que la définition générale d'une fonction en Python suit la syntaxe suivante :

Syntaxe : Définition de fonction

On peut définir de nouvelles fonctions qui seront ensuite utilisables comme les fonctions habituelles grâce à la syntaxe suivante :

```
def nomFonction(var1,var2,...):
    """Documentation de la fonction"""
    premiere ligne de code
    deuxieme ligne de code
    ...
    return ValeurDeRetour
```

On rappelle aussi que l'instruction `return` ***a un double effet*** :

- elle arrête (définitivement) l'exécution de la fonction ;
- elle renvoie la valeur de retour de la fonction. Cette valeur peut être une ***liste*** ou un ***tuple***, ce qui permet de renvoyer plusieurs valeurs en même temps au besoin.

Ex. 4.1 Écrire une fonction `recherche(liste,valeur)` qui renvoie l'indice de la première occurrence de `valeur` dans la `liste` si elle existe, et qui renvoie la longueur de la liste sinon.

Ex. 4.2 Écrire une fonction `maximum(liste)` qui renvoie la valeur maximale de la liste `liste` ainsi que l'indice de la première occurrence de cette valeur.

Ex. 4.3 Écrire une fonction `cherche(chaine,mot)` qui cherche si `mot` est présent dans `chaine` (ces deux paramètres étant des chaînes de caractères) et renvoie l'indice de la première lettre de sa première occurrence si c'est le cas, ou renvoie la longueur de `chaine` si `mot` n'est pas présent.

Il existe par ailleurs une seconde syntaxe de déclaration de fonction :

Syntaxe : Définition de fonction (bis)

On peut définir de nouvelles fonctions à l'aide d'une seconde syntaxe, plus compacte, mais moins puissante que la première :

```
nomFonction = lambda var1,var2,... : expression(var1,var2,...)
```

Ex. 4.4

1) Définir à l'aide de l'instruction `lambda` les fonctions `carre(x)`, `cube(x)`, `racinecarree(x)`.
2) Définir une fonction `approxDerivee(f,dx)` qui prend en paramètre ***une fonction*** `f` et un flottant `dx` et qui renvoie ***la fonction*** `fp` définie par `fp(x)=(f(x+dx)-f(x))/dx`.
3) Tester la fonction `approxDerivee` sur les trois fonctions définies à la question 1) pour différentes valeurs de `dx`. Retrouve-t-on des valeurs proches de la dérivée théorique de chacune de ces fonctions ?

I.2. Variables locales/variables globales

Lorsqu'une variable est ***déclarée à l'intérieur du corps d'une fonction***, elle ***n'est utilisable et accessible que dans cette fonction*** : on dit que la variable est ***locale***.
Au contraire, lorsqu'elle est déclarée en dehors du corps d'une fonction, elle est utilisable et accessible partout : on dit que la variable est ***globale***.

```
>>> x=1
>>> def exemple():
...     x=2
...     n=-1
...     print(x,n)
...
>>> exemple()
2 -1
>>> print(x)
1
>>> print(n)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Dans cet exemple, la fonction `exemple` affecte la valeur 2 à la ***variable locale*** `x`, puis affiche la valeur de cette variable. ***La variable étant locale***, ceci n'a aucune influence sur ***la variable globale*** `x`, dont la valeur est restée égale à 1, même après que la fonction ait été exécutée.
On peut désirer ***déclarer une variable à l'intérieur du corps d'une fonction mais la rendre accessible partout dans le code***. Pour cela, on utilise l'instruction `global`.

Syntaxe : `global`

L'instruction `global` permet de déclarer une variable dans le corps d'une fonction en la rendant utilisable et accessible hors de cette fonction. Sa syntaxe est la suivante :

```
def maFonction:
    global mavariable
    ...
```

```
>>> x=-5
>>> def initialisation():
...     global x,n
...     x=1.
...     n=0
...
>>> initialisation()
>>> print("x vaut",x,"et n vaut",n)
x vaut 1.0 et n vaut 0
```

Ici, l'instruction `global` a permis de rendre les variables `x` et `n` globales, bien qu'elles soient définies ou modifiées dans le corps de la fonction. De ce fait, les modifications apportées à ces variables à l'intérieur de la fonction ont une influence sur la valeur de ces variables en dehors de la fonction.
D'une manière générale, l'usage de variables globales est fortement déconseillé : le fait que les variables utilisées dans le corps d'une fonction soient par défaut locales est le comportement naturel et souhaité pour une fonction.
Ex. 4.5 Quels sont les effets des codes suivants :

1)
```
def fonc():
    a=2
fonc()
print(a)
```

2)
```
def fonc():
    global a
    a=2
fonc()
print(a)
```

3)
```
a=1
def fonc():
    a=2
fonc()
print(a)
```

4)
```
a=1
def fonc():
    global a
    a=2
fonc()
print(a)
```

Cor. 4.5

I.3. Rôle du `print` et du `return` dans une fonction

D'une manière générale, l'utilisation de `print` ***dans une fonction doit être proscrite. On utilisera systématiquement*** `return` ***dès que la valeur de retour de la fonction est connue***.

Il y a deux exceptions à cette règle :

- certaines fonctions ont pour but d'afficher un résultat, et non pas de renvoyer une valeur. Pour ces fonctions, on utilisera `print` afin d'afficher ce que l'on souhaite, et on n'utilisera pas de `return` puisque ces fonctions n'ont pas pour but de calculer une valeur.
- lorsqu'une fonction ne renvoie pas la valeur désirée (suite à une erreur de programmation par exemple), on peut momentanément insérer quelques `print` dans le corps de la fonction pour chercher où se situe l'erreur. Dès que l'erreur est trouvée, on enlève ces `print`.

Ex. 4.6

1) Écrire une fonction `X(n)` qui prend pour argument un ***entier*** `n` et a pour effet d'***afficher*** les $2n+1$ lignes décrites par les exemples suivants :

```
>>> X(1)
* *
 *
* *
```

```
>>> X(2)
*   *
 * *
  *
 * *
*   *
```

etc...

2) Écrire une fonction `A(n)` qui prend pour argument un ***entier*** `n` et a pour effet d'***afficher*** les $2n+1$ lignes décrites par les exemples suivants :

```
>>> A(1)
  *
 ***
*   *
```

```
>>> A(2)
    *
   * *
  *****
 *     *
*       *
```

I.4. Paramètres nommés optionnels : exemple de la fonction `print`

La fonction `print` permet, nous l'avons déjà très souvent vu, d'afficher autant d'objet que voulu. ***Par défaut, lors de leur affichage, ces objets sont séparés par espaces, et à la fin de l'affichage on passe automatiquement à une nouvelle ligne***. Ce comportement par défaut de la fonction `print` peut être modifié.
Pour cela, on utilise deux paramètres optionnels de la fonction, qui pour pouvoir être modifiés doivent être appelés par leur nom :

- le paramètre `sep` permet de préciser quel caractère (ou chaîne de caractères) sera utilisé comme séparateur ;
- le paramètre `end` permet de préciser quel caractère (ou chaîne de caractères) sera utilisé comme caractère de fin de ligne.

Par défaut, `sep=' '` et `end='\n'`.

```
>>> for i in range(10):
...     print('7*',i,'=',7*i,sep='',end='; ')
...
7*0=0; 7*1=7; 7*2=14; 7*3=21; 7*4=28; 7*5=35; 7*6=42; 7*7=49; 7*8=56; 7*9=63;
```

Ex. 4.7 On rappelle que la fonction `chr(k)` permet d'obtenir le caractère associé à un entier k donné (dans le standard UTF8).
Écrire une fonction `caracteres(n)` permettant d'***afficher***, pour ***tous les entiers*** k ***de*** 0 ***jusqu'à*** $n-1$, ***sur une même ligne***, les différentes valeurs de k suivies de ' :' suivi du caractère correspondant, tout ça séparés par ' - '.

I.5. Paramètres nommés optionnels : utilisation dans une définition de fonction

Au moment de la définition de nouvelles fonctions, on peut décider que certains paramètres seront optionnels :

Syntaxe : Paramètres nommés optionnels d'une fonction

Pour déclarer un paramètre nommé optionnel, il suffit de lui affecter une valeur ***à l'intérieur de l'entête de la fonction***.
Lors d'un appel à la fonction, si la valeur du paramètre est omise il prendra par défaut la valeur à laquelle il a été affecté dans l'entête.
Au contraire, si l'on souhaite donner une valeur à ce paramètre, il faut utiliser la même syntaxe que pour les paramètres `sep` et `end` de la fonction `print`.
Exemple :

```
import numpy as np
def approximation(x,n=5):
    i=0; xr=x; lst=[]
    while i<n:
        lst=[int(np.floor(x))]+lst
        if abs(x-lst[0])<1e-10:
            break
        x=1/(x-lst[0])
        i+=1
    p,q=lst[0],1
    for i in range(len(lst)-1):
        p,q=p*lst[i+1]+q,p
```

```
    print("Une approximation rationnelle de",xr,"est : ",end="")
    if q==1: print(str(p))
    else: print(str(p),"/",str(q),"=",p/q,sep="")

approximation(np.sqrt(2))
Une approximation rationnelle de 1.4142135623730951 est : 41/29=1.4137931034482758
approximation(np.sqrt(2),n=8)
Une approximation rationnelle de 1.4142135623730951 est : 577/408=1.4142156862745099
```

Ex. 4.8 Modifier le code de la fonction `approxDerivee` de l'exercice 4.4 de sorte à ce que le paramètre soit optionnel et ait 10^{-7} pour valeur par défaut.
Obtenir à l'aide de cette fonction une valeur approchée de la dérivée de sin en $\frac{\pi}{3}$.

I.6. Exercices

Ex. 4.9 Écrire une fonction `vecteur(A,B)` qui prend en paramètre deux ***listes*** représentant respectivement les coordonnées de deux points A et B et renvoie une liste représentant les coordonnées du vecteur $\overrightarrow{AB}$.

Ex. 4.10 Écrire une fonction `milieu(A,B)` qui prend en paramètre deux ***listes*** représentant respectivement les coordonnées de deux points A et B et renvoie une liste représentant les coordonnées du milieu du segment $[AB]$.

Ex. 4.11 Écrire une fonction `angle(v1,v2)` qui prend en paramètre deux ***listes*** représentant respectivement les coordonnées de deux vecteurs $\vec{v_1}$ et $\vec{v_2}$ et renvoie l'angle (orienté) $(\vec{v_1};\vec{v_2})$.

Ex. 4.12 Écrire une fonction `angle(A,O,B)` qui prend en paramètre trois ***listes*** représentant respectivement les coordonnées de trois points A, O et B et renvoie l'angle (orienté) $\left(\overrightarrow{OA};\overrightarrow{OB}\right)$.

Ex. 4.13 Écrire une fonction `longueur(A,B)` qui prend en paramètres deux listes représentant les coordonnées des points A et B et renvoie la longueur AB.

Ex. 4.14 Écrire une fonction `translation(v)` qui prend en paramètre une liste représentant les coordonnées du vecteur $\vec{v}$ et renvoie ***la fonction*** qui à tout point M du plan (représenté par la liste de ses coordonnées) associe son image par la translation de vecteur $\vec{v}$.

Ex. 4.15 Écrire une fonction `homothetie(C,r)` qui prend en paramètre une liste représentant les coordonnées du point C et un flottant r et renvoie ***la fonction*** qui à tout point M du plan (représenté par la liste de ses coordonnées) associe son image par l'homothétie de centre C et de rapport r.

Ex. 4.16 Écrire une fonction `rotation(C,a)` qui prend en paramètre une liste représentant les coordonnées du point C et un flottant a et renvoie ***la fonction*** qui à tout point M du plan (représenté par la liste de ses coordonnées) associe son image par la rotation de centre C et d'angle a.

Ex. 4.17 Écrire une fonction `isobarycentre(famille)` qui prend en paramètre une ***liste de points*** (chaque point étant représenté par la ***liste*** de ses coordonnées) et renvoie les coordonnées de l'isobarycentre de cette famille.

II. Objets et modules

II.1. Classes et objets

Définition 4.1 (Classes et objets)

On appelle ***classes*** des types particuliers dont les variables sont appelées ***objets*** et dont les caractéristiques sont les suivantes :

- chaque objet d'une classe possède des variables internes appelées ***attribut*** décrivant les

propriétés de cet objet ;
- chaque objet d'une classe possède des fonctions internes appelées ***méthodes*** qui implémentent le fonctionnement de cet objet ou qui lui confèrent certaines capacités.

En pratique, en Python, ***tous les types sont des classes et toutes les variables sont donc des objets***.

II.2. Exemples et classes de référence

- ***La classe*** `complex` :

```
>>> c=3-2j
>>> c.conjugate() # méthode : parenthèses obligatoires. Renvoie le complexe conjugué
(3+2j)
>>> c.imag        # attribut : pas de parenthèses
-2.0
>>> c.real        # attribut : pas de parenthèses
3.0
```

- ***La classe*** `list` :

```
>>> L=[1,-19,3.5,0.]
>>> L.append(1)       # méthode. NE RENVOIE RIEN. Ajoute l'élément donné en paramètre
>>> #                 # en fin de liste.
>>> L.count(1)        # méthode. Renvoie le nombre d'occurrences du paramètre dans L
2
>>> L.sort()          # méthode. NE RENVOIE RIEN. Trie la liste depuis laquelle la
>>> #                 # méthode est appelée.
>>> L.index(3.5)      # méthode. Renvoie l'indice du premier élément de la liste égal
4
>>> #                 # au paramètre.
>>> L.insert(0,7.3)   # méthode. NE RENVOIE RIEN. Insère dans la liste le second
>>> #                 # paramètre, en utilisant l'indice donné dans le premier paramè
>>> L.pop()           # méthode. Renvoie le dernier élément de la liste tout en le
3.5
>>> #                 # supprimant de la liste.
```

- ***La classe*** `str` :

```
>>> S="abc123abc000"
>>> S.count('0')      # méthode. Renvoie le nombre d'occurrences du paramètre dans S
3
>>> S.index('3')      # méthode. Renvoie l'indice du paramètre si c'est une sous-chaîne
5
>>> #                 # de S. Renvoie une erreur sinon.
>>> ';'.join(list(S))# méthode. Renvoie la chaîne de caractères obtenue en concaténan
'a;b;c;1;2;3;a;b;c;0;0;0'
>>> #                 # les éléments de la liste donnée en paramètre, séparés par la
>>> #                 # depuis laquelle la méthode est appelée.
>>> S.split('abc')    # méthode. Renvoie la liste des chaînes obtenues en séparant S
['', '123', '000']
>>> #                 # avec comme séparateur le paramètre fourni.
```

II.3. Modules

Trois modules sont au programme d'informatique : `numpy`, `matplotlib`, et `scipy`.
En pratique d'autres modules peuvent être utiles comme `time`, `random`, `os`, `sys`, etc...
Chaque module est constitué d'un ensemble de nouvelles fonctions et classes enrichissant le noyau Python.
Plus précisément :

Définition 4.2 (Module)

Un ***module*** est une extension du langage Python apportant de nouveaux objets, fonctions, etc...
Nous avons déjà utilisé plusieurs modules : `os`, `math`, ...

Notation

Syntaxe d'import d'un module et accès aux objets importés
Pour charger un module, on utilisera de préférence la syntaxe suivante :

```
import nom_du_module as abreviation
```

Le module est alors accessible à l'aide de l'`abreviation` qui le désigne et les objets qui y sont définis grâce à la même syntaxe que pour les attributs et méthodes d'un objet.

```
abreviation.objet_du_module
```

Notamment, cette syntaxe pour l'import de modules permet d'utiliser l'auto-complétion (à l'aide de la touche de tabulation) et d'obtenir aisément tous les objets définis dans le module importé.

II.4. `matplotlib`

Le module `matplotlib` permet de faire divers types de représentations graphiques. Chargeons les modules `math` et `matplotlib` :

```
>>> import pylab
>>> import matplotlib as mpl
>>> import math as m
```

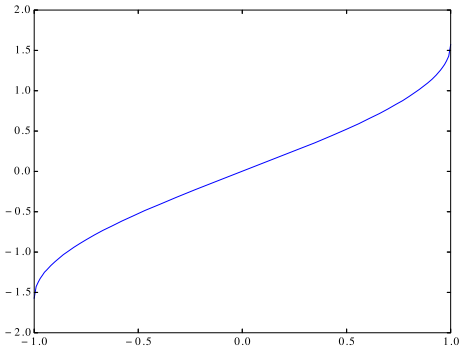
`mpl` désigne désormais le module `matplotlib` et `m` le module `math`.
Parmi les objets définis dans `matplotlib`, nous allons utiliser `pyplot` :

```
>>> RG=mpl.pyplot
```

II.5. `matplotlib.pyplot`

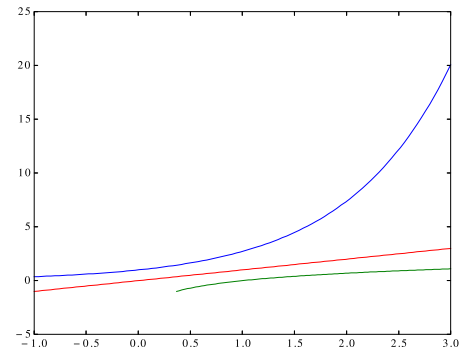
`RG` désigne désormais l'objet `matplotlib.pyplot`.
Pour représenter une fonction à l'aide de notre nouvel objet `RG`, il va falloir utiliser sa méthode `plot` avec pour paramètres la liste des abscisses de points de la représentation graphique suivie de la liste de leurs ordonnées.
Traçons par exemple la représentation graphique de la fonction Arcsin :

```
>>> nb_pts=200
>>> abscisses=[-1+2*k/(nb_pts-1) for k in range(nb_pts)]
>>> ordonnees=[m.asin(k) for k in abscisses]
>>> RG.plot(abscisses,ordonnees)
[<matplotlib.lines.Line2D object at 0x000001BB28B9CF08>]
```



Pour tracer plusieurs représentations graphiques dans un même repère, il suffit de faire plusieurs appels successifs à la méthode `plot`.
Traçons dans un même repère les représentations graphiques des fonctions $x \mapsto \exp(x)$, $x \mapsto \ln(x)$ et $x \mapsto x$.

```
>>> nb_pts=100
>>> abscisses1=[-1+4*k/(nb_pts-1) for k in range(nb_pts)]
>>> ordonnees1=[m.exp(k) for k in abscisses1]
>>> abscisses2=[m.exp(-1)+(3-m.exp(-1))*k/(nb_pts-1) for k in range(nb_pts)]
>>> ordonnees2=[m.log(k) for k in abscisses2]
>>> RG.plot(abscisses1,ordonnees1)
[<matplotlib.lines.Line2D object at 0x000001BB2C002C08>]
>>> RG.plot(abscisses2,ordonnees2)
[<matplotlib.lines.Line2D object at 0x000001BB2C00E308>]
>>> RG.plot(abscisses1,abscisses1)
[<matplotlib.lines.Line2D object at 0x000001BB2C00EB48>]
```

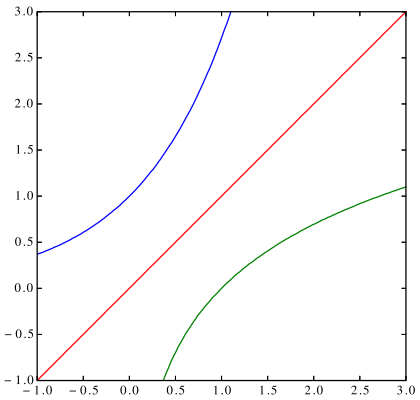


Comme on le voit, les représentations graphiques des fonction ln et exp qui sont censées être symétriques l'une de l'autre par rapport à la droite d'équation $y = x$ ne le sont pas.
C'est non seulement dû au mauvais choix des intervalles mais aussi à la ***nature du repère qui n'est pas orthonormé***.
Pour changer cela, on utilise la méthode `axis` de `RG` avec la syntaxe :

```
RG.axis('scaled') #obtenir un repere orthonorme
RG.axis([xmin,xmax,ymin,ymax]) #modifier les bornes de la fenetre graphique
```

```
>>> nb_pts=100
>>> abscisses1=[-1+(1+m.log(3.))*k/(nb_pts-1) for k in range(nb_pts)]
>>> ordonnees1=[m.exp(k) for k in abscisses1]
>>> abscisses2=[m.exp(-1)+(3-m.exp(-1))*k/(nb_pts-1) for k in range(nb_pts)]
>>> ordonnees2=[m.log(k) for k in abscisses2]
>>> abscisses3=[-1+4*k/(nb_pts-1) for k in range(nb_pts)]
```

```
>>> RG.plot(abscisses1,ordonnees1)
[<matplotlib.lines.Line2D object at 0x000001BB2C01D608>]
>>> RG.plot(abscisses2,ordonnees2)
[<matplotlib.lines.Line2D object at 0x000001BB2C01DD48>]
>>> RG.plot(abscisses3,abscisses3)
[<matplotlib.lines.Line2D object at 0x000001BB2C01DD08>]
>>> RG.axis('scaled');RG.axis([-1.,3.,-1.,3.])
(-1.2, 3.2, -2.6536129892940252, 21.168353585686795)
[-1.0, 3.0, -1.0, 3.0]
```



Le module `matplotlib` et son objet `pyplot` ont bien d'autres fonctionnalités que nous continuerons à découvrir tout au long de l'année.