TP de Python numéro 2 - Fonctions, boucles while

*Semaine du **26 Septembre**.*

I. Le module `numpy`

Commentaires en programmation

Tout langage de programmation un peu abouti permet de faire des commentaires. Il s'agit juste de vous permettre de commenter votre code pour :

- le rendre plus lisible ou expliquer le rôle d'une ligne,
- ou rendre tout un bout de code inactif ce qui est souhaitable dans plusieurs situations (par exemple, si on veut débugger son code).

En `Python`, le symbole # indique un début de commentaire sur une ligne, et les triples guillemets `"""` délimitent une zone commentée.
Copier coller le code ci-dessous et le lancer, que fait la fonction `np.sqrt` ?

```
import numpy as np #Le bloc commenté ci-dessous explique cette ligne

'''
Cette ligne importe le module numpy qui rajoute des fonctions mathématiques à Python,
comme la fonction sqrt.
Testons ce qu'elle fait à l'aide du code ci-dessous
'''

for i in range(1,20):
    print("pour i valant ", i ," np.sqrt(i) renvoie ", np.sqrt(i))
```

Lors de vos TPs, vous utiliserez les triples guillemets pour garder une trace de vos travaux, sans faire relire à `Python` le travail des exercices précédents à chaque fois.

1. Le module `numpy`

Remarque. Dans le TP de la semaine dernière, nous avons vu des fonctionnalités de base de `Python`, auxquelles il convient d'ajouter la fonction valeur absolue `abs` et la notation scientifique : tapez `1.23e4` et `Python` interprétera `1.23*10**4`.

Si `Python` fournit un cadre de travail dans lequel on peut utiliser les opérations de base des mathématiques, on souhaite aussi s'en servir pour des objets plus compliqués. Par exemple, la fonction exponentielle n'est pas, par défaut, implémentée en `Python`. Mais ce langage est très utilisé en sciences, il serait surprenant que personne n'ait déjà eu envie d'utiliser l'exponentielle...

Les **modules** d'un langage de programmation (ici `Python`) permettent d'enrichir ce langage, notamment en ajoutant des fonctions.

Chaque module vient avec sa documentation, qui explique comment est codée chaque fonction ajoutée. Par exemple, pour implémenter l'exponentielle, on pourrait utiliser la formule (voir semestre 2) suivante :

$$\mathrm{e}^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

valable pour tout réel x, et pour adapter cela à l'ordinateur - qui ne calcule jamais de sommes infinies - on

pourrait apprendre à `Python` à utiliser l'approximation :

$$e^1 \simeq \sum_{k=1}^{100} \frac{1}{k!}.$$

Le module `numpy` (*numbers for Python*) est le module le plus utilisé pour faire des mathématiques.

Pour s'en servir dans son code, il faut :

1. Le télécharger (avec la commande `pip install numpy` ou `install numpy` selon votre version de `Python`). Normalement, c'est déjà fait (il suffit de le faire une fois sur votre machine).
2. L'appeler au début de votre programme, avec la commande `import numpy`.
3. Les fonctions de `numpy` sont alors disponibles, mais il faut toujours dire à `Python` d'aller les chercher dans `numpy` en écrivant par exemple `numpy.exp` pour la fonction exponentielle `exp` ajoutée par numpy.

En fait, on utilise souvent un raccourci pour gagner du temps.

Importer un module et le renommer

Pour importer le module `numpy`, après l'avoir bien téléchargé, on utilise en début de programme la ligne suivante :
`import numpy as np`
où:
- `import` demande à `Python` d'importer un module,
- `numpy` est le nom du module qu'on veut importer,
- `as` demande à `Python` de renommer ce module,
- `np` est le nom (personnalisable) qu'on décide de donner au module.

Après cette ligne, la fonction exponentielle `exp` de `numpy` est accessible en tapant :
`np.exp`

Exercice 1. Importer `numpy` à l'aide de la syntaxe précédente. Comprendre, en testant sur l'invite de commande, ce que donnent les commandes suivantes de `numpy` (le noter ici):

1. `np.pi`
2. `np.e`
3. `np.sqrt`
4. `np.exp`
5. `np.log`
6. `np.log10`
7. `np.floor`
8. `np.ceil`

Exercice 2. Coder un programme demandant à l'utilisateur d'entrer un nombre réel x, et qui :

1. Affiche "Entrez un nombre strictement positif" si le nombre donné est négatif.
2. Sinon, le programme doit afficher (de manière lisible) la racine carrée et le logarithme népérien de x, puis une phrase en français donnant la plus grande des deux quantités précédentes.

II. Les fonctions

1. La notion de "renvoi" et de fonction

La notion de renvoi est importante pour comprendre ce qui se passe lorsque `Python` procède à la lecture de votre code. Considérons par exemple que pendant cette lecture, au cours d'une ligne, `Python` doive effectuer un calcul comme `2+2`. Il va alors :

- interpréter 2+2 comme un calcul qu'il doit effectuer,
- effectuer ce calcul et noter le résultat, 4, puis

- continuer à lire sa ligne *comme si le résultat 4 avait été écrit à la place de 2+2.*

On dit alors que le calcul $2+2$ **renvoie** la valeur 4. La valeur renvoyée par une opération va "prendre sa place" une fois l'opération effectuée.

Un autre exemple : à la lecture de la ligne
`x=2+x`
`Python` va successivement:

- Commencer par comprendre `x=` comme une affectation de variable. Pour continuer, il sait qu'il attend une valeur (à mettre dans la variable `x`).
- Il continue sa lecture : il lit donc l'entier `2` suivi du signe `+`. Il sait alors qu'il va devoir opérer une addition, et doit continuer à lire pour savoir quelle somme effectuer.
- Il lit ensuite la lettre `x`. Il l'interprète ici comme un nom de variable. Il va donc chercher dans sa mémoire...
  - si la variable `x` n'est pas encore attribuée, `Python` s'arrête et affiche une erreur ("Qui est `x` ?").
  - Si la variable `x` est bien attribuée, il va chercher sa valeur v, et le symbole `x` lu ici va **renvoyer** cette valeur.
- Dans le second cas, où `x` a renvoyé une valeur v, `Python` doit donc effectuer le calcul $2+v$. Si cela a un sens, il le fait et ce calcul `renvoie` alors la valeur V trouvée.
- Enfin, `2+x` ayant renvoyé V, la ligne est interprétée finalement comme `x=`V, et `Python` met donc cette valeur V dans la variable `x`.

En programmation, **la notion de fonction** est capitale. Elle permet d'écrire un sous programme et de s'en servir facilement dans la suite. Par exemple, regardons le code suivant.

```
def reste2(x):
        return x%2
```

- Une fois ce code lu par `Python`, une nouvelle fonction nommée `reste2` est définie.
- Elle prends un argument, ici noté `x`,
- et renvoie le résultat du calcul `x%2` (rappel : il s'agit du reste de x dans sa division euclidienne par 2).

Autrement dit, quand `Python` lira `reste2(34)`, il ira chercher le bloc indenté précédent, l'interprétera avec `x=34`, et ici continuera sa lecture comme s'il avait lu `34 %2` .
Autrement dit, la fonction `Reste2` appliquée à une donnée `x` **renvoie** la donnée `x%2`. Dans `Python`, le mot clé `return` indique à `Python` ce qu'il devra renvoyer, dans le cadre d'une fonction.

2. Définir une fonction

Voici un autre exemple :

```
def fun(n,k):
        for i in range(n):
                print(i)
        if k < n:
                return k
        else:
                return n
```

Définir une fonction en python

Pour définir une fonction :
- On annonce la création d'une fonction avec `def`,
- on choisit un nom pour la fonction, ici `fun`,
- on spécifie le nombre de ses arguments et on les nomme. Ici, il y a deux arguments nommés `k` et `n`.
- Une fois ceci fait, on utilise les deux-points et on indente le code correspondant à `fun`.
- Dans notre exemple, la fonction commence par afficher tous les entiers de `0` à `n-1`.
- Ensuite, la fonction **renvoie** l'argument `k` si le test `k<n` donne `True`, et renvoie `n` sinon.

Quand une fonction renvoie une valeur avec `return`, `Python` considère que le travail de la fonction est terminé et **cesse de lire** le code de la fonction. `return` a donc un rôle tout a fait particulier. On peut très bien définir une fonction sans argument, ou sans `return`.

Les arguments d'une fonction sont aussi appelées ses **entrées**. Lorsque `Python` applique une fonction à une valeur donnée, on dit qu'il fait un appel de la fonction ("call" en anglais, pour la lecture d'erreur).

Exercice 3. On veut une fonction `f` qui, étant donné deux nombres x et y, renvoie la donnée $(x+1)(y-1)$. Déterminer les erreurs de syntaxe dans la définition de fonction ci-dessous.

```
def f(x,y)
        a=x+1
        b=y-1
return a*b
```

3. Exercices

Exercice 4. Définir en python la fonction `c` donnée, pour x un réel convenable, par $c(x) = \left(1+\frac{1}{x}\right)\left(1-\frac{1}{1+x}\right)$.
Faire calculer des valeurs de c, émettre une conjecture et la démontrer.

Exercice 5. Soient a et b deux réels strictement positifs. On appelle :

- Moyenne **arithmétique** de a et b le réel $\frac{a+b}{2}$,
- moyenne **géométrique** de a et b le réel $\sqrt{ab}$
- moyenne **harmonique** de a et b le réel $\frac{2}{\frac{1}{a}+\frac{1}{b}}$.

Coder trois fonctions `ma,mg,mh` prenant en entrée deux réels supposés strictement positifs et renvoyant respectivement leurs moyennes arithmétique, géométrique et harmonique.

Exercice 6. En utilisant les fonctions de l'exercice précédent, comparer (au sens de $\leq$) ces trois notions de moyenne pour beaucoup de valeurs de a et b. Vérifier un résultat démontré en TD.

Exercice 7.

1. Compléter le code suivant pour qu'il définisse une fonction d'entête `def estCarre(n):` prenant en entrée un entier `n` et renvoyant `True` si `n` est le carré d'un entier, et `False` sinon.
```
def estCarre(n):
        reponse=False
        for k in range(n+1):
                if n... :
                        reponse=True
        return(reponse)
```
2. Même question, avec une structure différente.
```
def estCarre(n):
        for k in range(n+1):
                if n... :
                        return(...)
        return(...)
```
3. Coder une fonction d'entête `def test7(n):` prenant en entrée un entier `n` et renvoyant `True` si $\texttt{n}^2+1$ est le carré d'un entier, et `False` sinon.
4. Soit n un entier. A quelle condition n^2+1 est-il le carré d'un entier? Conjecturer une réponse à l'aide de la question précédente.
5. Démontrer par l'absurde votre conjecture.

Exercice 8.

1. Écrire une fonction prenant en entrée deux entiers naturels `n` et `p` et renvoyant la valeur de la somme :
$$\sum_{k=0}^{n} k^p.$$
2. Á l'aide de cette fonction, émettre une conjecture reliant les sommes $\sum_{k=0}^{n} k$ et $\sum_{k=0}^{n} k^3$, pour tout entier n.
3. (Maths) En déduire une formule simple donnant $\sum_{k=0}^{n} k^3$ en fonction de $n \in \mathbb{N}$ puis la démontrer.

4. Exercices avec des listes

Rappel : Si `L` est une liste :

- `len(L)` renvoie la longueur de la liste `L`
- Les éléments de `L` sont numérotés à partir de 0, donc de 0 à `len(L) -1`.
- `L[i]` renvoie le `i`-ième élément de `L` si `i` est un entier de $[\![0, \texttt{len(L)} - 1]\!]$, et renvoie une erreur *"list index out of range"* sinon.
- On peut boucler sur les éléments de `L` avec : `for e in L:`.
- La commande `L.append(a)` rajoute l'élément `a` à la fin de la liste `L`.

Exercice 9. Tester et comprendre le code suivant. Quelle est la différence majeure entre ces deux boucles?

```
L=[1,2,4]
print("taille : ", len(L))

for x in L:
        print(x)
print("fin première boucle")

for i in range(len(L)):
        print("Le ",i,"ième élément de L est : ", L[i])
print("fin seconde boucle")
```

Exercice 10. Coder une fonction d'entête `def Moyenne(L):` prenant en entrée une liste de nombres `L` et renvoyant en sortie la moyenne des éléments de `L`.

Exercice 11. Coder une fonction d'entête `def GardePositifs(L):` prenant en entrée une liste de nombres `L` et renvoyant en sortie une nouvelle liste `G` constituée des éléments positifs de `L`.

On pourra, dans ce code, initialiser une variable de type liste `G`*, vide, en écrivant :*
`G=[]`
et rajouter les éléments de `L` *à* `G` *seulement si ceux-ci sont positifs.*

Remarque. Prenez l'habitude de tester systématiquement vos fonctions sur quelques exemples pour vérifier qu'il n'y a pas d'erreur.

Exercice 12. Tester la fonction de l'exercice précédent sur au moins 2 listes de nombres de votre choix.

III. Les boucles while

En plus des boucles `for`, on peut faire exécuter à répétition un bloc de code à l'aide d'une boucle `while`. Voici un exemple:

```
def fonction1(x):
        n=0
        while n<x:
                n=n+1
        return(n)
```

Cette fonction :

- Prends en entrée un argument noté `x`. Cette fonction n'a de sens que si `x` est un nombre (flottant ou entier par exemple).
- Initialise une variable locale `n` à 0.
- Répète, tant que `n<x`, l'opération `n=n+1` (remarquez l'indentation). On dit que la variable `n` est incrémentée de 1 à chaque tour de boucle.
- Une fois la boucle `while` terminée (rupture d'indentation), la fonction renvoie `n`

Ainsi, la fonction `fonction1` prends en entrée un nombre, et renvoie le plus petit entier positif supérieur à ce nombre.

La syntaxe générale d'une boucle `while` est la suivante :

```
while CONDITION:
        ligne à exécuter 1
        ligne à exécuter 2
        etc...
suite du code à effectuer après la fin de la boucle
```

où `CONDITION` doit renvoyer un booléen. Les lignes à lire à chaque tour de boucle sont, encore une fois, indentées.

Attention aux boucles while

Il est très facile de faire une boucle while ne s'arrêtant jamais. Au cas ou cela vous arrive, vous devez interrompre l'exécution de python (cherchez "interrupt the current running code" sur votre IDE). Pour que cela n'arrive pas, il faut avant tout que votre condition `CONDITION` puisse voir sa valeur de vérité changer au fur et à mesure que la boucle se répète.

Par exemple, la boucle suivante affichera "Bonjour" à tout jamais si on ne l'arrête pas.

```
n=0
while n<2:
        print("Bonjour")
        n=n-1
```

En effet, `n` est initialisé à 0 et, à chaque tour de boucle, se voit diminué de 1, donc `n` ne dépassera jamais 2 : la condition `n<2` sera toujours vraie.

Exercice 13. Que renvoie la fonction `mystere` suivante, prenant en entrée un nombre `x` ?

```
def mystere(x):
        if x>=0:
                n=0
                while n+1<=x:
                        n+=1 #n+=1 a le même effet que n=n+1
                return(n)
        else:
                while x<n:
                        n+=-1 #Plus généralement, n+=a a le même effet que n=n+a
                return(n)
```

Remarque. Les boucles `while` sont très utiles quand on ne sait pas à l'avance le nombre de tour de boucle dont nous aurons besoin.

Exercice 14.

1. Coder une fonction d'entête `def puissancemax(x):` prenant en entrée un entier strictement positif `x` et renvoyant le plus grand entier `n` tel que $2^{\texttt{n}} \leq \texttt{x}$.
2. Considérons la suite $u = (u_n)_{n\in\mathbb{N}}$ définie par la relation de récurrence :

$$\begin{cases} u_0 = 1 \\ \forall n \in \mathbb{N}, u_{n+1} = 3u_n - 1 \end{cases}.$$

   Coder une fonction d'entête `def rangmax(x):` prenant en entrée un nombre `x` et renvoyant le plus petit rang `n` pour lequel $u_{\texttt{n}} > \texttt{x}$. On admet que la suite u tend vers $+\infty$, ce qui assure que cet entier existe quelque soit l'entrée `x`.

La suite de Syracuse ci-dessous est au coeur d'un problème toujours irrésolu. On ne sait pas démontrer que, quelque soit la valeur de son premier terme, elle finit toujours par retomber sur les valeurs $1, 4, 2, 1, 4, 2, \ldots$ (si la suite retombe sur 1, alors elle prendra ces valeurs 1 puis 4 puis 2 indéfiniment : elle sera *périodique à partir d'un certain rang*).

Exercice 15. Soit $A \in \mathbb{N}$. On appelle suite de Syracuse de premier terme A l'unique suite u telle que $u_0 = A$ et vérifiant la relation de récurrence :

$$\forall n \in \mathbb{N}, u_{n+1} = \begin{cases} \dfrac{u_n}{2} \text{ si } u_n \text{ pair} \\ 3u_n + 1 \text{ sinon} \end{cases}.$$

1. Coder une fonction d'entête `def Syr(A,n):` prenant en entrée deux entiers naturel `A` et `n` et renvoyant en sortie la valeur du `n` ième terme de la suite de Syracuse de premier terme A.

2. Coder une fonction d'entête `def testSyr(A):` prenant en entrée un entier `A` et renvoyant en sortie le premier rang n pour lequel la suite de Syracuse de premier terme `A` vaut 1. *Si ça devait ne jamais arriver, le code attendu tournerait indéfiniment.* On n'utilisera pas la fonction de la question précédente, pour des raisons d'efficacité algorithmique.

IV. Affectations multiples

En `Python`, on peut affecter simultanément plusieurs variables. Par exemple :

```
a,b=1,2
```

a le même effet que :

```
a=1
b=2
```

Cela est de plus simultané. Par exemple,

```
a,b=b,a
```

inverse le contenu des variables `a` et `b` (si elles sont définies au préalable).

Exercice 16. Quelle est la différence entre la ligne précédente, et les deux lignes suivante?

```
a=b
b=a
```

Voici un exemple d'utilisation classique de l'affectation multiple.

Exercice 17. On considère la suite u donnée par

$$\begin{cases} u_0 = 1, u_1 = 2 \\ \forall n \in \mathbb{N}, u_{n+2} = 2u_{n+1} + u_n \end{cases}.$$

1. Calculer à la main le 5e terme u_5 de cette suite.
2. Compléter le code suivant pour qu'il affiche u_{100}.

```
u,v=1,2
for k in range(...):
        u,v=v,...
print(...)
```

Exercice 18. On considère une suite u définie à l'aide d'une relation de récurrence sur deux rangs :

$$\begin{cases} u_0 = a, u_1 = b \\ \forall n \in \mathbb{N}, u_{n+2} = u_n e^{-u_{n+1}} \end{cases},$$

où a et b sont des réels.

1. Écrire une fonction Python d'entête `def U(a,b,n):` prenant en entrée les valeurs `a` et `b` de a et b ainsi qu'un entier n, et renvoyant en sortie le terme $u_{\mathtt{n}}$ de la suite obtenue pour ces premiers termes.
2. Vérifier que pour $(a, b) = (1, 2)$, on a $u_{15} \simeq 1.697197256756246$. *En fait, Python ne calcule que des approximations, car les valeurs renvoyées par la fonction exponentielle de numpy sont approximatives !*
3. Sans utiliser la fonction `U` de la première question, coder une fonction d'entête `def ListeU(a,b,n)` prenant en entrée les mêmes arguments et renvoyant en sortie la liste

$$[\mathtt{u}_0, \mathtt{u}_1, \ldots, \mathtt{u}_\mathtt{n}].$$

Vérifier que `ListeU(2,3,5)` renvoie :

$$[2, 3, 0.09957413673572789, 2.7156685113435852, 0.006587895488026882, 2.697836772319214].$$