

# TD1 : Itération et récursion

## Logiciels de programmation Python

On utilisera dans ces TDs le langage python. De nombreux logiciels existent pour programmer en python, parmi lesquels Pyzo et Jupyter, qui sont notamment utilisés dans les concours de recrutement de l'éducation nationale (CAPES et Agrégation).

Sur les ordinateurs du bâtiment Mirande, Spyder est généralement installé et extrêmement similaire à pyzo.

Vous êtes encouragés à installer spyder, pyzo, et/ou jupyter sur votre ordinateur personnel pour travailler cette matière, et à défaut, sur les ordinateurs du bâtiment Mirande, vous pouvez lancer spyder en tapant `spyder3` dans un "terminal", tandis que jupyter s'exécute en tapant `/opt/anaconda3/bin/jupyter-notebook`.

Les TDs de ce cours sont disponibles à l'adresse <http://leurent.perso.math.cnrs.fr/lmo6b1/>. En particulier des fichiers python sont directement fournis pour vous éviter de recopier l'énoncé. Le fichier <http://leurent.perso.math.cnrs.fr/lmo6b1/TD1.ipynb> est utilisable avec Jupyter, tandis que le fichier <http://leurent.perso.math.cnrs.fr/lmo6b1/TD1.py> est utilisable avec spyder ou pyzo.

Tandis que jupyter produit des documents esthétiquement plaisants à regarder, spyder (ou pyzo) présente l'avantage d'une interface plus simple d'utilisation, et de fonctionnalités supplémentaires (comme l'explorateur de variables). Vous êtes donc fortement encouragés à privilégier l'utilisation de pyzo ou spyder (plutôt que jupyter) dans ce cours.

## Boucle for (Pour) : 1er exemple

Le code ci dessous calcule, pour les valeurs successives  $i=1$ ,  $i=5$ , puis  $i=9$ , le nombre  $i^2$ , puis l'imprime. Après cela, il affiche leur somme.

```
somme=0          # "somme" va désigner la somme partielle
for i in [1,5,9]: # valeurs prises par "i"
    k=i**2        # calcul de  $i^2$ 
    somme=somme+k # on ajoute un terme à la somme partielle
    print(k)      # impression de  $k=i^2$ 
print("La somme vaut",somme)
```

À noter :

\* Pour exécuter ce code (que ce soit avec spyder, pyzo, ou jupyter) tapez Ctrl+Entrée lorsque le curseur est sur le code. Python exécute alors l'ensemble de "la cellule". Python exécutera, dans une autre partie de l'écran sur le côté, tout le code jusqu'à la prochaine ligne commençant par "`%%`".

\* Vous pouvez directement entrer des commandes dans l'éditeur, et créer de nouvelles cellules en commençant une ligne par "`%%`".

\* La fonction "`print`" imprime une information puis fait un retour à la ligne

\* L'indentation (c'est à dire l'espacement au début de chaque ligne) permet à python de savoir que `print(k)` est à l'intérieur de la boucle (il le répète à chaque fois), alors que `print("La somme vaut",somme)` est à l'extérieur de la boucle (il n'est exécuté qu'une fois)

\* Le calcul de  $i^2$  se fait en tapant `i**2`.

\* Après chaque occurrence d'un `#`, le reste de la ligne est complètement ignoré par python qui ne l'exécute pas. Il s'agit donc de commentaires destinés à une personne qui lirait le fichier.

## 2ème exemple

```
for i in "Salut ": print (i)
print("tout le monde!")
```

À noter :

\* `for i in "Salut "` signifie que l'on répète la boucle pour chaque élément de "Salut ". Les différents éléments sont les caractères (les lettres et l'espace).

\* Dans le cas très particulier où il n'y a qu'une commande à l'intérieur de la boucle, python accepte qu'on écrive cette commande sur la même ligne que `for`, après les "`:`". Il n'est toutefois pas recommandé de prendre cette habitude qui produit rapidement des erreurs quand on imbrique les boucles les unes dans les autres.

\* La dernière commande `print("tout le monde!")` est à l'extérieur de la boucle.

## Remarques sur les listes et les chaînes de caractères

Chaque objet python a un "type", par exemple exécutez le code ci-dessous pour avoir le type de 3.2, de 8, de "Salut", de {1,5,9} et de [1,5,9]

```
for i in [3.2,8,'Salut',{1,5,9],[1,5,9]]:
    print(i,"est de type",type(i))
```

Par exemple `[1,5,9]` est une liste, c'est à dire un objet constitué de plusieurs éléments. Lorsque `i` vaut `[1,5,9]`, on peut accéder au premier élément en exécutant `i[0]` :

```
print("Si i=",i,", alors i[0] vaut ",i[0])
```

De même le deuxième élément s'appelle `i[1]`, le troisième s'appelle `i[2]`, etc  
De plus le dernier élément s'appelle aussi `i[-1]`, l'avant dernier s'appelle `i[-2]` etc

```
print("et i[-2] vaut ",i[-2])
```

on peut coller bout à bout les listes, comme ci-dessous :

```
print("[3,5,8]+[9,12,4]=", [3,5,8]+[9,12,4])
```

## Exemples de manipulations

les chaînes de caractères se manipulent comme des listes :

```
print("Salut"[3])
```

Pour les listes comme les chaînes de caractères, on peut extraire des sous-listes : par exemple

```
k=[1,5,9,2,6,4,8,2,1,5,9]  
print(k[2:5])
```

où `k[2:5]` désigne la sous liste `[k[2],k[3],k[4]]` (c'est à dire qu'on commence à l'indice 2 et qu'on s'arrête juste avant l'indice 5)

```
print("bonjour"[:-1])
```

## Conversions

Il est aisé de convertir des objets d'un type vers un autre, comme ci dessous :

```
print(set([2,8,5]),list({2,8,5}),float(15),int("15")+2)
```

par exemple `set([2,8,5])` a converti la liste `[2,8,5]` en l'ensemble `{2,8,5}` (qui peut aussi s'écrire `{2,5,8}` ou toute autre permutation)

## Exercice

**Calculer la somme**  $\sum_{k=0}^5(k^3 + k - 1)$

Astuce : on peut utiliser la fonction `range` pour produire la liste de valeurs `[0,1,2,3,4,5]`. N'hésitez pas à consulter l'aide pour plus de précisions.

```
t=0  
for k in .....  
    t=t+.....  
print(t)
```

**À vous de remplir les "....." avec des instructions pertinentes**

## Autres forme de boucles for : utilisation d'une liste

On peut demander d'enregistrer dans une liste un calcul effectué plusieurs fois : par exemple on calcule ci dessous les  $k^3 + k - 1$  pour  $0 \leq k \leq 5$

```
valeurs=[k**3+k-1 for k in range(6)]  
print("les valeurs sont", valeurs)
```

on peut alors calculer la somme de la question précédente :

```
print("leur somme vaut",sum(valeurs))
```

## Autres forme de boucles for : utilisation d'un générateur

On peut informer python qu'on s'intéresse à  $k^3 + k - 1$  pour  $0 \leq k \leq 5$ , sans lui demander immédiatement de calculer ces valeurs :

```
valeurs=(k**3+k-1 for k in range(6))
```

Dans ce cas python ne calcule pas à l'avance toutes les valeurs, mais il peut les calculer l'une après l'autre : calculez `next(valeurs)` plusieurs fois de la console pour voir ce qu'il se passe

De même on peut calculer `sum(valeurs)` (ou `sum(k**3+k-1 for k in range(6))`) qui est équivalent)

## Boucles while (Tant que)

Le code ci-dessous affiche le plus petit entier naturel  $k$  tel que  $k^2 - k - 8 > 0$

```
k=0
while k**2-k-8<=0:
    k=k+1
print("k=",k, " est le plus petit entier naturel k tel que k**2-k-8>0" )
```

Explication :

il commence par  $k=0$ , et à chaque itération :  
il vérifie si  $k^2 - k - 8 > 0$ . Si ce n'est pas le cas, il ajoute 1 à  $k$  et recommence l'itération  
si  $k^2 - k - 8 > 0$ , alors il cesse la boucle (la condition dans "while" n'étant pas satisfaite)  
une fois la boucle finie, la variable  $k$  a gardé en mémoire la valeur qu'elle avait à la dernière itération, c'est à dire la valeur pour laquelle on a eu  $k^2 - k - 8 > 0$ . On décide alors de l'afficher.

**Question : Peut-on trouver de la même façon le plus petit entier naturel  $k$  tel que  $k^2 - k + 8 < 0$  ?**

## Instructions "continue" et "break" (Arrêt)

Cette fois-ci, on cherche uniquement parmi les entiers naturels plus petits que 12, quel est le plus petit tel que  $k^2 - k - 8 > 0$

```
k=0
while k<12:
    if k**2-k-8<=0:
        k=k+1
        continue
    print("k=",k, "" est le plus petit des entiers naturels k<12
        tel que k**2-k-8>0"" )
    break
```

Explication

on commence à  $k=0$  est on effectue une boucle, où à chaque itération :  
si  $k >= 12$  on cesse d'itérer (c'est ce que dit l'instruction while  $k < 12$ )  
si  $k^2 - k - 8 \leq 0$ , on veut réessayer avec la valeur suivante. Donc on fait "k=k+1", puis "continue" qui signifie qu'on passe à l'itération suivante sans exécuter la suite de la boucle

sinon on exécute la suite : on affiche que l'on a trouvé la valeur de  $k$ , et on sort de la boucle (la commande "break" signifie qu'on arrête d'itérer)

## Exercice

Le programme suivant est sensé trouver le minimum de la fonction  $n^2 - 2n + 2$  (définie pour  $n$  entier)

```
k=0
valeur=k**2-2*k+2
while True:
    if k>1000:
        print("pas de minimum trouvé "+
            "parmi les 1000 premières valeurs")
        break
    k=k+1
    nouvelle_valeur=k**2-2*k+2
    if nouvelle_valeur<valeur:
        valeur=nouvelle_valeur
        continue
    print("le minimum est",valeur,", qui est atteint pour n=",k-1)
    break
```

Déterminez (avec un papier et un crayon) ce qu'il se passe quand on exécute ce code. (vous pourrez ensuite vérifier en l'exécutant sur l'ordinateur)

Cela détermine t'il bien le minimum de la fonction  $n^2 - 2n + 2$  (définie pour  $n$  entier) ?

## Définition de fonctions

Ci-dessous, on définit ci dessous la fonction  $f : \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ k & \mapsto & k^2 - k - 8 \end{cases}$  puis on affiche  $f(8)$

```
def f(k):
    if type(k)==int:
        return k**2-k-8
    else: raise ValueError ("La fonction f n'accepte que des entiers")
print(f(8))
```

Remarque : "==" test si deux objets sont égaux, alors que "=" enregistre une valeur dans une variable

**Question :**

**Que se passe t'il si on calcule f(3.2) ?**

## Fonctions récursives

Ci-dessous, on définit de manière récursive la fonction factorielle, puis on affiche la factorielle de 37

```
def factorielle(n):  
    if type(n)!=int or n<0: raise ValueError ("""La fonction  
        factorielle n'est définie que sur les entiers naturels""")  
    elif n==0:return 1  
    else: return n*factorielle(n-1)  
  
print(factorielle(37))
```

À Noter : "elif" signifie "else if".

**On pourrait imaginer de la définir plus simplement par**

```
def factorielle(n): return n*factorielle(n-1)
```

**Pourquoi cela ne conviendrait-il pas ?**

## Exemple : suite de Fibonacci

La suite de Fibonacci est définie par :

$$F_0=F_1=1$$

$$\forall n \geq 1 : F_{n+1}=F_n+F_{n-1}$$

Les deux programmes suivants la calculent de manière récursive

```
def fibonacci1(n):  
    if type(n)!=int or n<0: raise ValueError  
    elif n<=1:return 1  
    else: return fibonacci1(n-1)+fibonacci1(n-2)
```

```
def fibonacci2(n):
```

```
if n==0:return 1  
def derniers_fibo(n):  
    if n==1:return [1,1]  
    a,b=derniers_fibo(n-1)  
    return [b,a+b]  
return derniers_fibo(n)[1]
```

```
print(fibonacci1(5))  
print(fibonacci2(5))
```

**Questions :**

- **Expliquez le fonctionnement de fibonacci2 : en particulier que fait la fonction "derniers\_fibo" ?**
- **Quand on calcule fibonacci1(5), combien de fois (environ) la fonction fibonacci1 est-elle appelée ?**
- **Indication : pour s'en rendre compte, on pourra insérer une ligne print("F(",n,")") après def fibonacci1(n):**
- **En quoi la fonction fibonacci2 est-elle plus satisfaisante ? On pourra comparer le temps de calcul de fibonacci1(100) et fibonacci2(100)**
- **Écrire à l'aide d'une boucle for ou while, une fonction équivalente à fibonacci2**
- **Écrire de même une fonction récursive qui calcule les n+1 premiers éléments de la suite de fibonacci (par exemple fibonacci3(5) doit renvoyer [1, 1, 2, 3, 5, 8])**

## Permutations

Écrivez une fonction récursive qui liste toutes de permutations possibles d'un ensemble

Par exemple permutations(4,8,2) doit renvoyer [[4, 2, 8], [2, 4, 8], [2, 8, 4], [4, 8, 2], [8, 4, 2], [8, 2, 4]] (éventuellement dans un autre ordre)

## Décomposition de 2019

Le code ci-dessous détermine le plus petit entier naturel  $k$  tel qu'il existe  $l \in \{1,2,3,\dots,k-1\}$  tel que  $k \times l = 2019$

```
k=0
while not any([k*l==2019 for l in range(1,k)]):k=k+1
print(k)
```

**Questions :**

- Expliquez le fonctionnement de ce code.
- Quand k vaut 673, que vaut `[k*l==2019 for l in range(1,k)]` ? Était-il vraiment nécessaire de calculer tous les éléments de cette liste ?
- Récrire le code de sorte qu'il ne vérifie pas si `k*l==2019` pour chaque valeur de l, mais qu'au contraire il s'arrête dès qu'il obtient "True"

## Évaluation paresseuse

La fonction "any" est évaluée de façon paresseuse. Cela signifie que si on remplace le code précédent par

```
k=0
while not any(k*i==2019 for i in range(1,k)):k=k+1
print(k)
```

alors il ne commence pas par calculer la liste `[k*i==2018 for i in range(1,k)]` mais il cherche directement si un de ses éléments est "True", en essayant une par une les valeurs de i. Quand on dit qu'il est paresseux, c'est qu'il sait que dès qu'il trouve une valeur "True", il peut s'arrêter là sans rien calculer pour les autres valeurs.

**Écrire un exemple qui montre explicitement cette différence de comportement entre `any(...)` et `any(...)`**