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, qui est utilisé dans les concours de recrutement de l'éducation nationale (CAPES et Agrégation).

Si vous n'avez pas accès à Pyzo, vous pouvez aussi utiliser spyder à la place (qui est quasiment identique, et installé sur les ordinateurs du premier étage).

En salle A202, vous pouvez lancer pyzo en ouvrant une "Konsole" et en y tapant `/opt/anaconda3/bin/pyzo` (suivi de la touche Entrée)

Installer pyzo (si vous utilisez votre propre ordinateur)

suivre les instructions sur `http://www.pyzo.org/install.html` :

— sous linux : `apt-get install python3-pyqt4 python3-pip` suivi de `python3 -m pip install pyzolib pyzo`

— sous windows : installer le fichier `https://github.com/pyzo/pyzo/releases/download/v4.4.2/pyzo-4.4.2-win32.exe`

Au premier lancement, pyzo demande où est installé python. Une fois répondu il affiche une fenêtre "console" où s'exécute python.

Différentes fenêtres

Pyzo (ou spyder) contient plusieurs fenêtres

— La "console Python" où l'on peut exécuter des commandes. Si vous y tapez par exemple `1+1` (puis la touche entrée), Python répondra `2`.

— L'éditeur, qui est l'intérêt principal de pyzo (ou spyder). Téléchargez puis ouvrez dans pyzo le fichier `http://leurent.perso.math.cnrs.fr/lmo6b1/TD1.py`. Le fichier est constitué de commandes en langage Python, entrecoupées de commentaires, et séparées en blocs appelés cellules. Pour créer de tels blocs, il suffit d'introduire des lignes commençant par `#%%`. Les touches Ctrl+Entrée permettent d'exécuter un tel bloc d'un seul coup. L'exécution des commandes se fait encore dans la console Python.

— D'autres fenêtres apparaissent, le menu "tools" (dans pyzo) permet de choisir lesquelles afficher. Il est conseillé d'afficher la fenêtre qui s'appelle "workspace" dans pyzo (elle s'appelle "explorateur de variable" dans spyder), et la fenêtre "interactive help" ("inspecteur d'objets" dans spyder).

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 imprime "j'ai fini".

```
for i in [1,5,9]:
    k=i**2
    print(k)
print("J'ai fini")
```

À noter :

* Pour exécuter ce code avec spyder ou pyzo tapez Ctrl+Entrée lorsque le curseur est sur le code. Python exécutera, dans une autre partie de l'écran sur le côté, tout le code jusqu'à la prochaine occurrence de `"#%%"`.

* 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("J'ai fini")` 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`.

* Toute ligne commençant par `#` est considérée comme un commentaire, elle n'est pas exécutée contrairement au reste du code

* Ce qui est entre des guillemets est du texte, que Python n'interprète pas comme des commandes à exécuter. C'est une autre façon (moins standard) d'ajouter des commentaires, utilisée dans le fichier python associé à cette feuille d'exercices.

Dans ce cas, si le texte s'étale sur plus d'une ligne, il faut le mettre entre de triples guillemets.

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 range(.............
    t=t+.............
print(t)
```

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

Boucle for dans une liste

Le calcul précédent peut simplement s'écrire `sum([k**3+k-1 for k in range(6)])`

En effet, `[k**3+k-1 for k in range(6)]` calcule la liste des valeurs de k^3+k-1 où k varie de 0 à 5.

```
print(sum([k**3+k-1 for k in range(6)]))
```

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 k²-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 :

S'il n'avait existé aucun entier naturel k tel que k^2-k-8>0, que se serait-il passé en exécutant ce code ?

Instructions "continue" et "break" (Arrêt)

Cette fois-ci, on cherche le plus petit k, parmi les entiers naturels plus petits que 12, 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 k²-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$\leqslant$0, on veut réessayer avec le 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)

Question :

S'il n'avait existé aucun k, parmi les entiers naturels plus petits que 12, tel que k^2-k-8>0, que se serait-il passé en exécutant ce code ?

Définition de fonctions

Ci-dessous, on définit ci dessous la fonction $f : \begin{cases} \mathbb{N} & \to & \mathbb{N} \\ k & \mapsto & k^2-k-8 \end{cases}$ 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
$\forall n \geqslant 1 : F_{n+1}=F_n+F_{n-1}$
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 la fonction `fibonacci1` est-elle appelée ?**
*** 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 2018

Le code ci-dessous détermine le plus petit entier naturel k tel qu'il existe l$\in${1,2,3,...,k-1} tel que k×l=2018

```
k=0
while not any([k*l==2018 for l in range(1,k)]):k=k+1
print(k)
```

Questions :
***Expliquez le fonctionnement de ce code.**
***Quand `k` vaut 1009, que vaut `[k*l==2018 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==2018` 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==2018 for i in range(1,k)]):k=k+1
print(k)
```

alors il ne commence par 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 essayer les autres valeurs.
Écrire un code qui montre explicitement cette différence de comportement entre `any([...])` et `any(...)`