Chapitre 1 : Itération et récursion

Logiciels de programmation Python

On utilisera dans ce cours 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 sypder en tapant `spyder3` dans un "terminal", tandis que jupyter s'exécute en tapant `jupyter-notebook`.

Les différents chapitres de ce cours sont (ou seront) disponibles sur plubel à l'adresse `https://plubel-prod.u-bourgogne.fr/course/view.php?id=1173`. Chaque chapitre y est disponible à la fois dans un format directement utilisable avec Jupyter, et dans un format 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).

I. Itération

1. Boucle `for` (Pour) :

1.1) Exemple introductif

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)
```

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". Si vous utilisez Pyzo ou Spyder, 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 `"#%%"`.

Le code indiqué utilise la fonction "`print`", qui imprime une information puis fait un retour à la ligne.

Notez que c'est l'indentation (c'est à dire l'espacement au début de chaque ligne) qui 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)

Enfin, on retiendra que le calcul de i^2 se fait en tapant `i**2`, et qu'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.

1.2) Iterations avec d'autres types des données

Dans l'exemple ci-dessous

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

On remarque que `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).

On notera que dans cet exemple `print(i)` est à l'intérieur de la boucle (éxecuté à chaque itération), tandis que `print("tout le monde!")` est à l'extérieur de la boucle (éxecuté uniquement à la fin).

Dans cet exemple, comme 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 peut être source d'erreurs quand on imbrique les boucles les unes dans les autres.

1.3) Exemple

Calculer la somme $\sum_{k=0}^{5}(k^3+k-1)$ en remplissant ci-dessous les "....." avec des instructions pertinentes

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

Astuce : on poura utiliser la fonction `range` pour produire la liste de valeurs `[0,1,2,3,4,5]`. N'hésitez pas à executer `?range` pour demander de l'aide sur la fonction `range`.

2. «Type» des données

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))
```

On notera aussi que, comme on peut le vérifier ci-dessous, python garde en mémoire la valeur qu'avait `i` lors de la dernière itération :

```
print("python se souvient que i vaut",i)
```

Sa valeur `[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)

3. Autres formes de boucles `for`

3.1) boucle `for` dans la définition 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 $0 \leqslant k \leqslant 5$

```
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))
```

3.2) boucle `for` définissant un «générateur»

On peut informer python qu'on s'intéresse à k^3+k-1 pour $0 \leqslant k \leqslant 5$, 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'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)

4. 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" )
```

En effet, il commence par `k=0`, et à chaque itération :

- il vérifie si $\texttt{k}^2 - \texttt{k} - 8 > 0$. Si ce n'est pas le cas, il ajoute `1` à `k` et recommence l'itération
- si $\texttt{k}^2 - \texttt{k} - 8 > 0$, 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 $\texttt{k}^2 - \texttt{k} - 8 > 0$. On décide alors de l'afficher.

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

5. Instructions "`continue`" et "`break`" (Arrêt)

5.1)

Pour illustrer les commandes `continue` et `break`, on écrit un code qui 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
else: print("il n'existe pas de k<12 tel que k**2-k-8>0" )
```

L'execution de ce code fait commencer à $\texttt{k} = 0$ puis effectuer une boucle, où à chaque itération :

- tant que $\texttt{k} < 12$ on execute le code à l'intérieur de la boucle suivante : (c'est ce que dit l'instruction `while k<12`)
  - si $\texttt{k}^2 - \texttt{k} - 8 \leqslant 0$, c'est que cette valeur de `k` n'est pas solution. On veut donc 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 completement d'itérer)
- si au moment de passer à l'itération suivante, il constate que `k` n'est plus inférieur à 12, alors il execute le `else` de la dernière ligne. Dans cet exemple cela ne peut survenir que si on a essayé chaque valeur sans jamais executer le `break`, c'est pourquoi le code demande alors qu'on affiche qu'il n'y a pas de solution

5.2) Exemple

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) ?

II. Récursion

1. Définition de fonctions

On définit ci dessous la fonction $f : \left\{ \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ k & \mapsto & k^2-k-8 \end{array} \right.$ 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))
```

La ligne `def f(k):` indique que l'on définit une fonction `f`, et que cette fonction prend un argument `k`

Après cette ligne l'indentation precise que les trois lignes suivantes forment la définition de `f` (c'est ce qui est executé quand on calcule `f(quelquchose)`), alors que la dernière ligne est à l'extérieur de la définition de `f`.

Notez enfin que `==` teste 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) ?

2. Fonctions récursives

Une fonction peut faire référence à elle même, de la même façon que certains objets mathématiques peuvent être définis par récurrence. Par exemple, on définit ci-dessous la fonction factorielle de manière récursive 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`". Noter aussi que dans cet exemple, cela ne changerait rien d'enlever le `else` et remplaçer le `elif` par un `if`.

On pourrait imaginer de la définir plus simplement par

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

Pourquoi cela ne conviendrait-il pas ?

III. Exercices

1. Décomposition de 2021

Le code ci-dessous détermine le plus petit entier naturel k tel qu'il existe $l \in \{1,2,3,\ldots,k-1\}$ tel que $k \times l = 2021$

```
k=0
while not any([k*l==2021 for l in range(1,k)]):k=k+1
print(k)
```

1. Expliquez le fonctionnement de ce code.
2. Quand `k` vaut 47, que vaut `[k*l==2021 for l in range(1,k)]` ? Était-il vraiment nécessaire de calculer tous les éléments de cette liste ?
3. Récrire le code de sorte qu'il ne vérifie pas si `k*l==2021` pour chaque valeur de `l`, mais qu'au contraire il s'arrête dès qu'il obtient "`True`"

2. 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]
```

Chacun de ces programmes permet par exemple de calculer F_5 :

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

1. Montrer que pour tout $n\in\mathbb{N}^*$, `derniers_fibo(n)` calcule le couple (F_{n-1},F_n). En déduire que pour tout $n\in\mathbb{N}$, `fibonacci2(n)` calcule F_n.
2. 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("appel de F(",n,")")` après `def fibonacci1(n):`
3. En quoi la fonction `fibonacci2` est-elle plus satisfaisante ? On pourra comparer le temps de calcul de `fibonacci1(100)` et `fibonacci2(100)`
4. Écrire de manière itérative (c'est à dire en utilisant une boucle `for` ou `while`, mais pas d'appel recursif), une fonction équivalente à `fibonacci2`
5. Écrire 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]`)

3. Suites définies par récurrence

3.1) Identification de suites

On définit les fonctions suivantes :

```
def f1(n):
    if n==0: return 2
    if n==1: return 6
    return 2*f1(n-1)+3*f1(n-2)

def f2(n):
    a=1
    for i in range(2,n): a=a*i
    return a

def f3(n):
    l=[2,6]+[0]*(n-1)
    for i in range(2,n+1):
        l[i]=2*l[i-1]+3*l[i-2]
    return l[n]
```

1. Pour chacune de ces fonctions, identifier une relation de récurence qu'elle satisfait, puis identifier la fonction mathématique calculée par le programme.
2. Déterminer le nombre d'itérations nécessaire au calcul de ces fonctions (ou un equivalent de ce nombre, ou au moins une majoration).

3.2) Calcul de suites fixées

On définit les suites suivantes :

$$\begin{cases} u_0 = 2 \\ \forall n \geqslant 1 : u_n = 3\,u_{n-1} \end{cases} \qquad \begin{cases} v_0 = 42 \\ \forall n\geqslant 0, (v_n = 0(mod 2)) \Rightarrow v_{n+1} = v_n/2 \\ \forall n\geqslant 0, (v_n = 1(mod 2)) \Rightarrow v_{n+1} = 3v_n + 1 \end{cases}$$

$$\begin{cases} \forall k\in\mathbb{Z}, w_{0,k} = \delta_{0,k} \\ \forall n\geqslant 1, \forall k\in\mathbb{Z}, w_{n,k} = w_{n-1,k-1} + w_{n-1,k} \end{cases}$$

1. Écrire pour chacune de ce suites une fonction python qui calcule sa valeur pour $n\in\mathbb{N}$ arbitraire.
2. Déterminer le nombre d'itérations nécessaire au calcul de ces fonctions (ou un equivalent de ce nombre, ou au moins une majoration).

4. Parties de $[\![0,n[\![$

Pour $n\in\mathbb{N}$, $\mathcal{P}([\![0,n[\![)$ designe l'ensemble de tous les sous-ensembles de $\{0,1,\ldots,n-1\}$.

1. Pour $n\in\mathbb{N}^*$, exprimer $\mathcal{P}([\![0,n+1[\![)$ à partir des éléments $\mathcal{P}([\![0,n[\![)$.
2. En déduire une fonction qui recursive qui calcule $\mathcal{P}([\![0,n[\![)$ pour $n\in\mathbb{N}^*$.
   Par exemple, pour `n`=2, cette fonction devra renvoyer `[{},{0},{1},{0,1}]` (ou éventuellement les mêmes ensembles dans un autre ordre).
   Pour simplifier vous pourrez renvoyer des listes à la place d'ensembles (les ensembles étant plus laborieux à manipuler en python).