

Programmation en Python -

Cours 3 : les listes

MPSI - Lycée Thiers

2014/2015

Les listes en python

Les listes

Création d'un itérateur avec range()

la boucle for

Boucle for

Exemples

Les listes

- Dès que l'on commence à manipuler en python un grand nombre de données l'usage de variable numérique devient insuffisant.

Exemple : imaginons que l'on veuille stocker les notes des élèves d'une classe pour calculer leur moyenne et leur écart-type. On peut imaginer d'utiliser N variables de type float (N étant le nombre d'élèves de la classe), puis d'écrire des fonctions `moyenne(.)` et `ecarttype(.)` prenant N paramètres ; fastidieux lorsque N=47, et il faudra la réécrire pour chaque nouvel effectif...

```
def moyenne47(n1, n2, ..., n47):  
    return (n1 + n2 + ... + n47) / 47
```

- En python la structure de donnée qui va nous aider est la *liste*. c'est un **objet** de type `list` qui permet de collecter des éléments : données de type quelconque : `int`, `float`, `str`, `bool`, ou d'autres listes, etc...

Les listes

Exemple :

```
>>> liste = [1, 2, 3, 'toto']
```

```
>>> print(liste)
[1, 2, 3, 'toto']
>>> type(liste)
<class 'list'>
```

```
>>> len(liste)
4
```

```
>>> liste[0], liste[1], liste[len(liste) - 1]
(1, 2, 'toto')
```

Une liste est créée à l'aide d'une affectation. Ses éléments sont entre crochets [.]. La fonction `len(.)` prend en argument une liste et retourne son nombre d'éléments. Les éléments d'une liste s'obtiennent grâce à leur **indice** entre crochets. Attention le premier élément a pour indice 0 !

Les listes

Exemple : reprenons notre problème de calcul de la moyenne de notes.
Mettons à profit ce que nous avons vu sur les listes ainsi que la fonction sum(.) qui prend en argument une liste et retourne, lorsque c'est possible, le résultat de l'opération '+' sur ses éléments.

```
>>> l = [10, 12, 14, 6, 8, 15, 3, 17]    # la liste de notes
```

```
>>> def moyenne(liste):                  # fonction moyenne(.)  
...     return sum(liste) / len(liste)  
...
```

```
>>> moyenne(l)  
10.625
```

Saisissons la liste des notes. Définition de la fonction `moyenne(.)` qui s'applique à toute liste non vide de nombres. On obtient le résultat attendu, la moyenne est de 10.625.

Les listes

Les listes sont des objets **modifiables** (on dit aussi **mutables**) : on peut modifier leurs éléments.

```
>>> liste = [1, 2, 3, 'toto']    # une liste
```

```
>>> liste[0] = 'le début'; liste[2] = [-1, -2, -3]
>>> print(liste)
['le début', 2, [-1,- 2, -3], 'toto']
```

```
>>> print(liste[-1], liste[-2])
'toto' [-1, -2, -3]
```

```
>>> print(liste[-5], liste[4])
... IndexError: list index out of range
```

On modifie un élément de la liste en lui affectant une nouvelle valeur (de n'importe quel type, simple ou complexe). L'indice -1 permet d'obtenir le dernier élément. C'est plus simple que `liste[len(liste) - 1]`. L'indice -2 l'avant-dernier, etc... Un indice qui n'est pas compris entre `-len(liste)` et `len(liste)-1` produit une erreur 'IndexError'.

Les listes

```
>>> print(liste)
['le début', 2, [-1,- 2, -3], 'toto']

>>> type(liste[0]), type(liste[1]), type(liste[2])
(<class 'str'>, <class 'int'>, <class 'list'>)

>>> print(liste[2][0])                # liste[2] est une liste
-1

>>> l=[[ 'a', 'b'], ['c', 'd']]
>>> print(l[0][0], l[0][1], '\n', l[1][0], l[1][1])
a b
c d
```

Les éléments d'une liste sont des valeurs de différents types, celles qu'on lui a affectées.

Une liste de listes permet de constituer un tableau bi-dimensionnel, etc...

Appartenance d'un élément à une liste : in

- La commande `in` permet de déterminer si un élément appartient ou non à une liste.

```
>>> liste = [1, -3, 5, 17.0, 2]
>>> 1 in liste
True
>>> -1 in liste
False
>>> -3.0 in liste
True
>>> 17 in liste
False
>>> 'toto' in liste
False
```


Parcours d'une liste : for Variable in Liste:

- Avec en plus la commande for on peut faire parcourir à une variable les éléments d'une liste :

```
>>> for i in liste:  
...     print(i)  
...  
1  
-3  
5  
17.0  
2
```

- Par exemple pour répéter 3 fois une instruction :

```
>>> liste = [0, 1, 2]  
>>> for i in liste:  
...     print('Bonjour')  
...  
Bonjour  
Bonjour  
Bonjour
```

La fonction range()

La dernière application du parcours d'une liste pour répéter une séquence d'instructions est intéressante. Mais couteuse en mémoire puisqu'une liste stocke en mémoire toutes les valeurs qu'elle contient.

- Pour cette raison (depuis la version 3 de python), la fonction `range(n)` retourne un *itérateur* sur les `n` premiers entiers naturels (de 0 à `n-1` inclus). Les valeurs intermédiaires ne sont plus stockées en mémoire. Cependant la commande `in` permet encore de déterminer l'appartenance d'un élément :

```
>>> print(range(10))
range(0,10)
>>> 2 in range(10)
True
>>> 10 in range(10)
False
```

- On peut convertir le résultat retourné en une liste grâce à la fonction de conversion `list()` :

```
>>> l = list(range(10)); print(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La fonction range()

- Avec deux arguments entiers m, n , `range(m,n)` retourne un itérateur sur les $\max(0, n - m)$ entiers consécutifs qui sont $m \leq . < n$.

```
>>> print(list(range(0,10))); print(list(range(-1,11)))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

- Avec trois arguments entiers m, n, k , `range(m, n, k)` retourne la liste des entiers de la forme $m+k*p$ avec p entier naturel qui sont compris entre m (inclu) et n (exclu).

```
>>> print(list(range(0,10,2))); print(list(range(10,0,-2)))  
[0, 2, 4, 6, 8]  
[10, 8, 6, 4, 2]
```

Ainsi : `range(n)` est équivalent à `range(0,n)` et à `range(0,n,1)`.



```
>>> list(range(0,10,-1))    # produit la liste vide  
[]
```

Par contre un argument non entier provoque un message d'erreur 'TypeError'.

Boucle for

L'instruction `for` permet de répéter une séquence d'instruction, en boucle, pour une variable qui décrit une liste (plus généralement un conteneur...) ou un itérateur d'éléments.

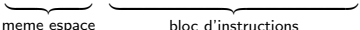
```
for variable in liste/itérateur:  
    ..... Instruction1  
    ..... Instruction2  
    ..... :  
    ..... Dernière instruction
```

 meme espace  bloc d'instructions

Boucle for

L'instruction `for` permet de répéter une séquence d'instruction, en boucle, pour une variable qui décrit une liste (plus généralement un conteneur...) ou un itérateur d'éléments.

```
for variable in liste      :  
    .....  Instruction1  
    .....  Instruction2  
    .....  :  
    .....  Dernière instruction
```



On l'écrit en langage algorithmique :

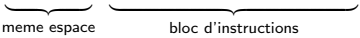
Pour tout i dans *liste* faire :
 Instruction 1
 :
 Instruction N

Bien sûr la variable `i` peut apparaître dans le bloc d'instruction (comme *variable locale* : c'est une 'copie', la modifier n'affecte pas la liste).

Boucle for

L'instruction `for` permet de répéter une séquence d'instruction, en boucle, pour une variable qui décrit une liste (plus généralement un conteneur...) ou un itérateur d'éléments.

```
for variable in range(N):  
    ..... Instruction1  
    ..... Instruction2  
    .....  
    ..... Dernière instruction
```



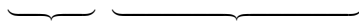
On l'écrit en langage algorithmique :

Pour tout i de 0 à $N-1$ faire :
 Instruction 1
 :
 Instruction N

Boucle for

L'instruction for permet de répéter une séquence d'instruction, en boucle, pour une variable qui décrit une liste (plus généralement un conteneur...) ou un itérateur d'éléments.

```
for variable in range(n,m,k):  
    ..... Instruction1  
    ..... Instruction2  
    .....  $\vdots$   
    ..... Dernière instruction
```


meme espace bloc d'instructions

On l'écrit en langage algorithmique :

Pour tout i de n à $m \pm 1$ par pas de k faire :
 Instruction 1
 \vdots
 Instruction N

+ ou - dépend respectivement de $k > 0$ ou $k < 0$.

Boucle for

```
for i in liste:
.....  instruction 1
.....  :
.....  instruction N
```

produit un résultat identique à la boucle while :

```
L = len(liste)          # L est la longueur de liste
j = 0                   # j est l'indice
while (j < L):          # tant que l'indice ne dépasse pas
.....  i = liste[j]     # i est l'élément d'indice j
.....  instruction 1
.....  :
.....  instruction N
.....  j = j + 1        # incrémenter l'indice j
```

lorsque instruction1,... ,instruction N ne modifient pas liste.

Une boucle while est plus générale. On a le droit de modifier la longueur de la liste dans la boucle while ; on n'en a pas le droit dans la boucle for (on peut toujours modifier les éléments de la liste). Dans une boucle for le nombre de répétition de la boucle est fixé à l'entrée dans la boucle; c'est `len(liste)`.

Exemples

Nous souhaiterions créer la liste des carrés des entiers compris entre 0 et 20. Pour cela on utilisera la **méthode** `list.append()` appliquée aux objets de type liste qui permet d'ajouter un élément en queue de liste :

```
>>> liste=[ ]; print(liste)
[]
>>> liste.append('toto'); print(liste)
['toto']
>>> liste.append('le héros'); print(liste)
['toto', 'le héros']
```

Solution :

```
>>> listeCarrés = [ ]          # initialisation
>>> for i in range(21):
...     listeCarrés.append(i ** 2)    # actualisation
...
```

```
>>> print listeCarrés
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
225, 256, 289, 324, 361, 400]
```

Exemples

Ecrire une fonction `fibonacci()` qui prend en argument un entier N et retourne une liste contenant les N premiers termes de la suite de Fibonacci :
 $u_0 = 0, u_1 = 1, \forall n \in \mathbb{N}, u_{n+2} = u_{n+1} + u_n.$

Solution :

```
>>> def fibonacci(N):  
...     result = [0, 1]           # Initialisation  
...     for k in range(2, N)  
...         result.append(result[k - 1] + result[k - 2])  
...     return result
```

La définition de la fonction est proche de la définition par récurrence de la suite, la boucle `for` jouant le rôle de la relation de récurrence.

```
>>> fibonacci(16)  
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

Exemples

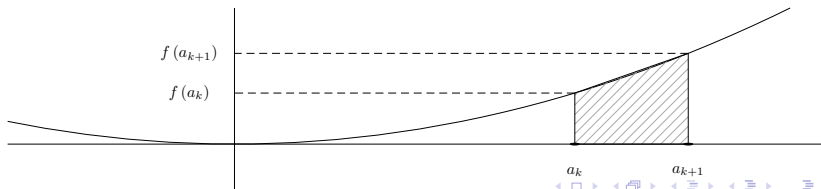
Soit l'application $f : x \mapsto x^2$ définie sur \mathbb{R}^2 .

Ecrire une fonction `trapeze(. , .)` qui prend en argument deux réels a et b et retourne une approximation de l'intégrale de f de a à b par la méthode des trapèzes. Comparer son résultat avec la valeur quasi-exacte.

Méthode des trapèzes : si $f : [a, b] \rightarrow \mathbb{R}$ est continue :

Soit $a_k = a + k \frac{b-a}{n}$ pour $k \in [[0, n]]$ (ainsi $a_0 = a$, $a_n = b$). Si n est assez grand :

$$\begin{aligned} \int_a^b f(x) dx &\approx \left(\frac{b-a}{n} \right) \sum_{k=0}^{n-1} \frac{f(a_k) + f(a_{k+1})}{2} \\ &\approx \left(\frac{b-a}{n} \right) \left(\frac{f(a) + f(b)}{2} + \sum_{k=1}^{n-1} f(a_k) \right) \end{aligned}$$



Exemples

```
def trapeze(a,b):  
    if a > b: return -trapeze(b,a)  
    N = 100  
    pas = (b-a)/N  
    res = (a**2+b**2)/2  
    for k in range(1,N):  
        res += (a + k*pas)**2  
    res *= pas  
    return res
```

la boucle for permet de calculer la somme.

```
>>> trapeze(0,1)      # intégrale de 0 à 1  
0.33335  
>>> 1**3 /3.         # valeur exacte  
0.3333333333333333  
>>> trapeze(0,100)   # intégrale de 0 à 100  
333350.0  
>>> 100**3 /3.       # valeur exacte  
333333.3333333333
```

Amélioration de la précision du dernier exemple

```
def trapeze(a,b):          # Version 2, meilleure précision
    if a > b: return -trapeze(b,a)
    N = int(100*(b-a))
    pas = (b-a)/N
    res = 0
    for k in range(1,N):
        res += (a + k*pas)**2
    res *= pas
    return res
```

la précision est meilleure :

```
>>> print('approchée:',trapeze(0,1), 'exacte:', 1**3/3)
approchée: 0.33335 exacte: 0.333333333333
>>> print('approchée:',trapeze(0,100), 'exacte:', 100**3/3)
approchée: 333333.335 exacte: 333333.333333
```