Chapitre (S1) 2 Listes

Objectifs

- Connaître l'objet *list* de python.
- Savoir manipuler les listes.
- Connaître les problèmes liés à la copie de listes.

1 NOTION DE LISTE

1.1 Qu'est ce qu'une liste ?

Une liste est une séquence (suite) d'objets qui peuvent être de types différents (y compris des listes). Une liste s'écrit entre deux crochets et ses éléments sont séparés d'une virgule. Chaque élément d'une liste est repéré par un indice entier.

1.2 Création d'une liste

On peut créer une liste vide de deux manières différentes :

```
>>> L1 = []
>>> L2 = list()
```

Une liste peut être définie en **extension** en écrivant tous ses éléments :

```
>>> L = [1, 2.5, [1, 2], 'a']
>>> type(L)
<class 'list'>
```

Une liste peut également être définie en **compréhension** en donnant une « formule » construisant la liste :

```
>>> L = [i**2 for i in range(10)]  # liste des carrés des entiers \
↪ <10
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Exercice 1 **Première construction** [Sol 1] Créer la liste les entiers pairs de 0 à 20 (exclu), en utilisant la construction par compréhension.

Remarque 1 La fonction `list(<objet>)` peut être utilisée pour créer une liste d'éléments à partir de n'importe quelle objet itérable[a] :

```
>>> L1 = list(range(1,10,2))
>>> L1
[1, 3, 5, 7, 9]
>>> L2 = list('Bonjour')
>>> L2
['B', 'o', 'n', 'j', 'o', 'u', 'r']
```

2 OPÉRATIONS ÉLÉMENTAIRES SUR UNE LISTE

2.1 Longueur d'une liste

Dans le cas d'une liste, la fonction `len()` renvoie l'entier égal au nombre d'objets présents dans cette liste, également appelé longueur de la liste

```
>>> L = [1, 2.5 ,[1, 2], 'A']
>>> len(L)
4
```

a. c'est-à-dire que l'on peut parcourir au moyen d'une boucle `for`

Lycée Michel MONTAIGNE – Bordeaux

2.2 Accès aux éléments

Les éléments d'une liste sont numérotés (on dit aussi indicés ou indexés) par des indices entiers. L'élément ℓ_i d'indice i d'une liste `L` s'écrit `L[i]`. Il existe deux sortes de numérotation (ou indexation).

- indexation par des entiers positifs : Les indices sont positifs et la numérotation débute à partir de 0. On peut représenter la liste `L` , contenant n éléments par

$$\texttt{L} = [\ell_0, \ell_1, \ldots, \ell_{n-1}]$$

Exemple :

```
>>> L = [1, 2.5 , [1, 2], 'A']
>>> L[1]
2.5
>>> L[2]
[1, 2]
```

- indexation par des entiers négatifs : Pour une liste à n éléments, `L[-1]` désigne le dernier élément, `L[-2]` l'avant dernier etc.
Exemple :

```
>>> liste = [1, 2.5 , [1, 2], 'A']
>>> liste[-1] # accès au dernier élément
'A'
>>> liste[-2] # l'avant-dernier
[1, 2]
```

Représentation schématique pour une liste de 5 éléments :

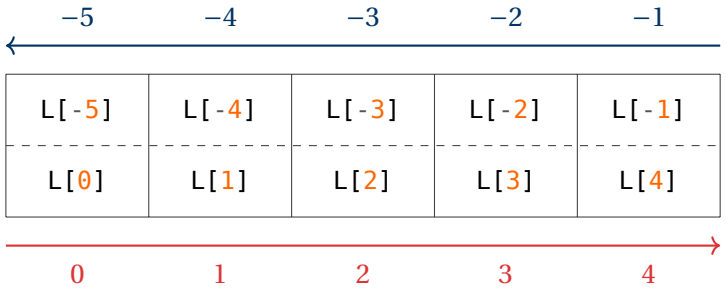

FIGURE 1 : Indexation des éléments d'une liste de longueur 5

Représentation schématique pour une liste de longueur quelconque :

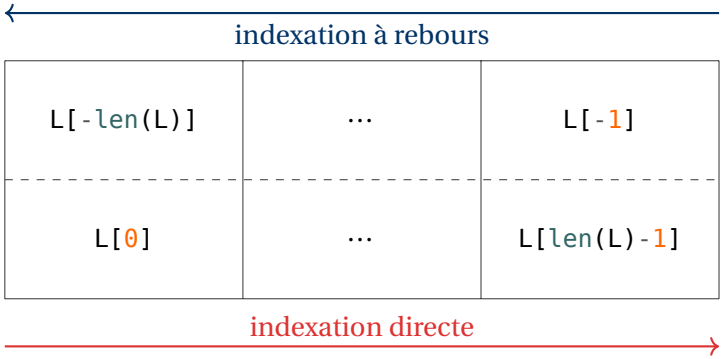

FIGURE 2 : Indexation des éléments d'une liste quelconque

2.3 Modification des éléments d'une liste

Les listes sont des objets **mutables** : on peut donc modifier les éléments individuellement :

```
>>> liste = [1, 2.5 , [1, 2], 'A']
>>> liste[1] = 3
>>> liste
[1, 3, [1, 2], 'A']
>>> liste[-1] = 'B'
>>> liste
[1, 3, [1, 2], 'B']
```

Si un élément d'une liste est une liste, on peut modifier de même ses éléments :

```
>>> liste = [1, 2.5 , [1, 2], 'A']
>>> liste[2][0] = 'un'
>>> liste
[1, 2.5, ['un', 2], 'A']
```

2.4 Parcours de liste

Il existe plusieurs manières de parcourir une liste.

- On peut utiliser un parcours par indice, à l'aide d'une boucle **`for`**, et d'une variable contenant les indices successifs.

```
>>> L = [1, 2.5 , [1, 2], 'A']
>>> for i in range(len(L)): print(i, L[i])
...
0 1
1 2.5
2 [1, 2]
3 A
>>> for i in range(-1,-len(L)-1,-1): print(i, L[i])
...
-1 A
-2 [1, 2]
-3 2.5
-4 1
```

- On peut aussi utiliser un parcours par élément, à l'aide d'une boucle **for**, et d'une variable contenant les éléments successifs de L. On dit que les listes sont des objets **itérables**[1].

```
>>> L = [1, 2.5 , [1, 2], 'A']
>>> for el in L: print(el)
...
1
2.5
[1, 2]
A
```

Remarque 2 On peut également utiliser `enumerate` :

```
>>> L = [1, 2.5 , [1, 2], 'A']
>>> for (i,el) in enumerate(L): print(i,el)  # i est \
↪ l'indice, el l'élément
...
0 1
1 2.5
2 [1, 2]
3 A
```

Exercice 2 Parcours de liste [Sol 2]

1. Créer une fonction `somme(L:list)->int` qui prend en argument une liste d'entiers et qui renvoie la somme des éléments de L, en utilisant un parcours par indice.

1. Dans ce cas, le parcours se fait obligatoirement dans le sens de lecture de la liste

2. Même question en utilisant un parcours par élément.

2.5 Concaténation, répétition

La concaténation (+) permet de mettre deux listes bout à bout pour en faire une nouvelle. La répétition (∗) permet de créer une liste en répétant un certain nombre de fois un ou plusieurs éléments.

```
>>> liste1 = ['lundi', 'mardi', 'mercredi']
>>> liste2 = ['jeudi', 'vendredi']
>>> liste3 = liste1 + liste2  # concaténation
>>> liste3
['lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi']
>>> liste4 = [0] *7  # répétition
>>> liste4
[0, 0, 0, 0, 0, 0, 0]
```

On peut ainsi construire une liste quelconque à partir d'une liste initialement vide en concaténant des listes à un élément.

Exemple 1 (liste des carrés des entiers <10)

```
L = []
for i in range(10):
    L = L+[i**2]
```

Exercice 3 Deuxième construction [Sol 3]

Créer la liste les entiers pairs de 0 à 20 (exclu), en utilisant la concaténation.

2.6 Comparaison & Appartenance

Le test d'égalité == entre listes est d'usage courant, il donne la valeur **True** si les deux listes ont les mêmes éléments dans le même ordre, **False** sinon.

La comparaison entre listes est possible mais n'est pas très employée. Elle est basée sur une comparaison *lexicographique*[2], et deux éléments distincts ne peuvent être comparés que s'ils sont de même type.

2. C'est-à-dire l'ordre du dictionnaire

```
>>> [1, 2] == ['3', 4]
False
>>> L = [1, 2]
>>> L == [1, 2]
True
>>> [1, 5] < [3, 4]  # ordre lexicographique
True
>>> [1, 2, 8] < [3]
True
>>> [1, 2] < ['3', 4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and \
↪ 'str'
```

Exercice 4 [Sol 4] Expliquez les résultats des appels précédents.

Pour savoir si une valeur figure dans une liste on utilise l'opérateur **in** :

```
>>> L = [1, 2.5 , [1, 2], 'A']
>>> 1 in L
True
>>> 2 in L
False
>>> [1, 2] in L
True
```

Remarque 3 L'instruction **in** n'est pas une opération élémentaire, car elle nécessite un parcours de liste, qui permet de tester chaque élément de la liste. Son temps d'exécution dépend du contenu de la liste.

3 OPÉRATIONS PLUS AVANCÉES SUR LES LISTES

3.1 Slicing

Pour extraire une sous-liste d'une liste L on utilise le « slicing », dont la syntaxe est :

`L[dep:fin:pas]`

où `dep`, `fin` et `pas` sont des entiers. Cette instruction permet d'extraire de L une sous-liste contenant les éléments dont les indices varient de `dep` (inclus) à `fin` (exclus), séparés de `pas`.

La partie `[:pas]` est facultative et si elle est omise, le pas vaut 1 par défaut. Si l'argument `dep` est omis, alors il vaut 0 par défaut (début de la liste), si l'argument `fin` est omis, alors c'est jusqu'à la fin de la liste. Si le pas est négatif il faut raisonner de droite à gauche.

Exemples :

```
>>> L = [1, 2.5 , [1, 2], 'A','B'] # définition de la liste
>>> L[1:4:2] # de L[1] à L[3] par pas de 2
[2.5, 'A']
>>> L[:3]  # les 3 premiers
[1, 2.5, [1, 2]]
>>> L[1:]  # tout à partir du deuxième
[2.5, [1, 2], 'A', 'B']
>>> L[1::2]  # de 2 en 2 à partir du deuxième
[2.5, 'A']
>>> L[-3:]  # les 3 derniers
[[1, 2], 'A', 'B']
>>> L[::] # toute la liste
[1, 2.5, [1, 2], 'A', 'B']
>>>
>>> L[3:0:-1] # du 4ième au 2ième en sens inverse
['A', [1, 2], 2.5]
```

Si `pas` est positif (resp. négatif), alors il faut que `dep` soit strictement inférieur (resp. supérieur) à `fin`, sinon la liste renvoyée est vide.

Exemples :

```
>>> L = [1, 2.5 , [1, 2], 'A','B'] # définition de la liste
>>> L[3:4] # seul l'élément L[3]
['A']
>>> L[3:3]  # liste vide
[]
>>> L[3:2:-1]  # seul l'élément L[3]
['A']
>>> L[3:3:-1]  # liste vide
[]
```

L'instruction **del** permet la suppression d'un élément (ou d'une tranche) connaissant son indice :

```
>>> L = [0, 1, '3/2', 2, 3, 4]
>>> del L[5]
>>> L
[0, 1, '3/2', 2, 3]
>>> del L[:2] # suppression des 2 premiers
>>> L
['3/2', 2, 3]
```

Il est bon de savoir réaliser « à la main » quelques modifications élémentaires de listes.

Exercice 5 **Modifications de listes** [Sol 5] On donne pour chaque question une liste `L` pré-remplie. Dans chaque cas, écrire une instruction qui modifie `L` pour parvenir à la liste `L = [1,2,3,4,5]` (plusieurs réponses possibles).

1. `L = [1,2,3]`
2. `L = [3,4,5]`
3. `L = [1,2,5]`
4. `L = [1,2,3,4,5,6,7]`
5. `L = [-1,0,1,2,3,4,5]`
6. `L = [1,2,3,3.5,4,5]`. Que donne `L[3] = []` ?

3.2 Méthodes associées aux listes

On les trouve en tapant dans la console `dir(list)`, ce qui donne :

```
>>> dir(list)
['__add__', '__class__', '__class_getitem__', '__contains__', \
↪ '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', \
↪ '__format__', '__ge__', '__getattribute__', '__getitem__', \
↪ '__getstate__', '__gt__', '__hash__', '__iadd__', '__imul__', \
↪ '__init__', '__init_subclass__', '__iter__', '__le__', \
↪ '__len__', '__lt__', '__mul__', '__ne__', '__new__', \
↪ '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', \
↪ '__rmul__', '__setattr__', '__setitem__', '__sizeof__', \
↪ '__str__', '__subclasshook__', 'append', 'clear', 'copy', \
↪ 'count', 'extend', 'index', 'insert', 'pop', 'remove', \
↪ 'reverse', 'sort']
```

On obtient ainsi la liste des méthodes applicables aux listes. Certaines de ces méthodes seront utilisées par la suite, il convient donc de connaitre leur syntaxe.

3.3 Méthodes append et insert

Pour ajouter un objet en fin de liste il y a la méthode `append(<objet>)` :

```
>>> L = [1, 2, 3]
>>> L.append(4)
>>> L
[1, 2, 3, 4]
```

Pour insérer un objet à l'indice *i* il y a la méthode `insert(<indice>,<objet>)`

```
>>> L = [1, 2, 5]
>>> L.insert(2, 4)
>>> L
[1, 2, 4, 5]
>>> L.insert(2, 3)
>>> L
[1, 2, 3, 4, 5]
```

Remarque 4 Comme vu précédemment, pour insérer une sous-liste à l'indice *i* on utilise le « slicing » :

```
>>> L = [1, 2, 5]
>>> L[2:2] = [3, 4]  # insertion à l'indice 2
>>> L
[1, 2, 3, 4, 5]
```

On notera bien la différence avec l'instruction `L.insert(2,[3,4])`, qui conduit à la liste `[1,2,[3,4],5]`. Ainsi, le principe du « slicing » permet de faire des insertions/suppressions en remplaçant la liste extraite (à gauche de l'affectation) par une autre sous-liste (à droite de l'affectation) :

```
>>> L = [1, 2, 2.5, '11/4', 4, 5]
>>> L[2:4] = [3]
>>> L
[1, 2, 3, 4, 5]
```

Exercice 6 **Troisième construction** [Sol 6] Créer la liste les entiers pairs de 0 à 20 (exclu), en utilisant la méthode `append`.

3.4 Méthodes pop et remove

La méthode `pop()` renvoie le dernier élément d'une liste, cet élément est alors supprimé de la liste :

```
>>> L = [1, 2, 3, 4]
>>> L.pop()
4
>>> L
[1, 2, 3]
```

On voit dans l'exécution précédente, que l'instruction `liste.pop()` renvoie quelque chose. On peut le stocker dans une variable si besoin.

```
>>> L = [1, 2, 3, 4]
>>> x = L.pop()
>>> x
4
```

La méthode `remove(<valeur>)` permet de suppxrimer la première occurrence d'une `<valeur>` dans une liste :

```
>>> L = [1, 2, 'trois', 3, 4, 'trois']
>>> L.remove('trois')  # suppression valeur d'indice 2
>>> L
[1, 2, 3, 4, 'trois']
```

Exercice 7 [Sol 7]

1. **[Mixage de listes]** Écrire la fonction `mixer(c1: list, c2: list)->list` qui renvoie la liste obtenue en mixant les deux arguments de la manière suivante : `[c1[0],c2[0],c1[1],c2[1],...]`, on s'arrête lorsque la liste la plus courte a été épuisée.

2. **[Comptage]** Étant donné une liste L d'entiers compris entre 0 (inclus) et 100 (exclu), écrire la fonction : `compter(L: list)->list` qui renvoie la liste des nombres d'entiers de la liste L compris dans l'intervalle $[10k; 10(k+1)[$ pour k allant de 0 à 9.

3. **[Moyenne]** On considère une liste de taille n contenant uniquement des nombres $L = [\ell_0, \ell_1, \dots, \ell_{n-1}]$. La moyenne des nombres de la liste est donnée par $\overline{m} = \frac{1}{n}\sum_{k=0}^{n-1} \ell_k$. Écrire une fonction `moy(L: list)->float` qui renvoie la valeur de la moyenne des éléments de la liste.

4. **[Variance]** On définit de même la variance de cet ensemble de nombre par $v = \frac{1}{n}\sum_{k=0}^{n-1}(\ell_k - \overline{m})^2$. Écrire une fonction `var(L: list)->float` qui renvoie la valeur de la variance des éléments de la liste.

4 COPIE D'UNE LISTE

4.1 Origine du problème

Une affectation ne duplique pas l'objet `list` (essentiellement pour des raisons d'occupation mémoire) mais crée une deuxième étiquette sur cet objet[3]. Les listes étant des objets mutables, ceci peut donner lieu à certains comportements surprenants lors de copies. En effet, si on a bien :

```
>>> a = 1
>>> b = a
>>> a, b
(1, 1)
>>> a = 2
>>> a, b
(2, 1)
```

on obtient de manière plus surprenante (au premier abord) :

```
>>> L1 = [1, 2, 3, [1, 2, 3]]
>>> L2 = L1
>>> L1[0] = 0
>>> L1[3][0] = 0
>>> L1
[0, 2, 3, [0, 2, 3]]
>>> L2
[0, 2, 3, [0, 2, 3]]
```

Comme on peut le constater, la modification de L1 a été répercutée sur la variable L2. En effet, les éléments de L1 n'ont pas été dupliqués, c'est l'adresse mémoire du contenu de L1 qui a été dupliquée dans L2, donc L1 et L2 pointent vers un même contenu en mémoire.

3. Une étiquette est un nom que l'on donne à un objet, c'est le nom qui est à gauche du symbole d'affectation (=). Concrètement, cela représente une adresse mémoire.

4.2 Comment dupliquer une liste ?

Une première idée pour dupliquer une liste est d'extraire toute la liste, mais le problème se retrouve au niveau des éléments qui sont eux-même des listes, ils ne seront pas dupliqués. Pour dupliquer totalement une liste, on utilise la commande `deepcopy` du module `copy`. On obtient alors :

```
>>> L1 = [1, 2, 3, [1, 2, 3]]
>>> L2 = L1
>>> L3 = L1[:]
>>> from copy import deepcopy
>>> L4 = deepcopy(L1)
>>> L1[0] = 0
>>> L1[3][0] = 0
>>> L1
[0, 2, 3, [0, 2, 3]]
>>> L2
[0, 2, 3, [0, 2, 3]]
>>> L3
[1, 2, 3, [0, 2, 3]]
>>> L4
[1, 2, 3, [1, 2, 3]]
```

On peut visualiser la gestion mémoire lors des affectations à l'aide de la figure ci-dessous :

FIGURE 3 : Visualisation d'affectations sur le site pythontutor

SOLUTIONS DES EXERCICES

Solution 1

```
>>> L1 = [i for i in range(0,20,2)]
>>> L1
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> L2 = [i for i in range(0,20) if i%2 == 0]
>>> L2
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Solution 2

1. Parcours par indice :

```
def somme(L:list)->int:
    S = 0 # variable qui contient la somme des éléments
    for i in range(len(L)):
        S = S+L[i]
    return S
```

2. Parcours par élément :

```
def somme(L:list)->int:
    S = 0 # variable qui contient la somme des éléments
    for e in L:
        S = S+e
    return S
```

Solution 3

Première méthode avec test.

```
L = []
for i in range(20):
    if i%2 == 0:
        L = L+[i]
```

Deuxième méthode avec parcours des entiers pairs uniquement.

```
L = []
for i in range(0,20,2):
    L = L+[i]
```

Solution 4

- ligne 2 : renvoie `False` car `1` est différent de `'3'`.
- ligne 5 : renvoie `True` car `L` et `[1, 2]` sont de même longueur et ont les mêmes éléments.
- ligne 7 : renvoie `True` car $1 < 3$.
- ligne 9 : renvoie `True` car $1 < 3$.
- ligne 10 : renvoie une erreur car on ne peut comparer un entier (`1`) avec une chaîne (`'3'`).

Solution 5

1. `L = L+[4,5]` (ou bien `L += [4,5]`)
2. `L = [1,2]+L`
3. `L[2:2] = [3,4]`
4. `L[5:] = []` (ou bien `del L[5:]`)
5. `L[:2] = []` (ou bien `del L[:2]`)
6. `L[3:4] = []` (ou bien `del L[3:4]`). L'instruction `L[3] = []` conduit à `[1,2,3,[],4,5]`

Solution 6

Première méthode avec test.

```
L = []
for i in range(20):
    if i%2 == 0:
        L.append(i)
```

Deuxième méthode avec parcours des entiers pairs uniquement.

```
L = []
for i in range(0,20,2):
    L.append(i)
```

Solution 7

1. On part d'une liste vide à laquelle on ajoute successivement un caractère de c1 suivi d'un caractère de c2 :

```
def mixer(c1: list, c2:list)->list:
    """mixer les listes c1 et c2"""
    # longueur m de la plus petite liste
    if len(c1) < len(c2):
        m = len(c1)
    else:
        m = len(c2)
    # mixage des listes
    s = []  # pour le résultat
    for i in range(m):
        s.append(c1[i])
        s.append(c2[i])
    return s
>>> c1 = ['a', 'b', 'c', 'd']
>>> c2 = [1, 2, 3, 4, 5, 6]
>>> c3 = mixer(c1, c2)
>>> c3
['a', 1, 'b', 2, 'c', 3, 'd', 4]
```

On peut aussi (mais moins joli) ajouter l'élément de tête de chaque liste, puis le supprimer.

```
def mixer(c1: list, c2:list)->list:
    """mixer les listes c1 et c2"""
    # longueur m de la plus petite liste
    if len(c1) < len(c2):
        m = len(c1)
    else:
        m = len(c2)
    # mixage des listes
    s = []  # pour le résultat
    for i in range(m):
        s.append(c1[0])
        del c1[0]
        s.append(c2[0])
        del c2[0]
    return s
>>> c1 = ['a', 'b', 'c', 'd']
>>> c2 = [1, 2, 3, 4, 5, 6]
>>> c3 = mixer(c1, c2)
>>> c3
['a', 1, 'b', 2, 'c', 3, 'd', 4]
```

Ou encore faire un test de parité avec une boucle `for` de longueur $2m$:

```
def mixer(c1: list, c2:list)->list:
    """mixer les listes c1 et c2"""
    # longueur m de la plus petite liste
    if len(c1) < len(c2):
        m = len(c1)
    else:
        m = len(c2)
    # mixage des listes
    s = []  # pour le résultat
    for i in range(2*m):
        if i%2 == 0:
            s.append(c1[0])
            del c1[0]
        else:
            s.append(c2[0])
            del c2[0]
    return s
>>> c1 = ['a', 'b', 'c', 'd']
>>> c2 = [1, 2, 3, 4, 5, 6]
>>> c3 = mixer(c1, c2)
>>> c3
['a', 1, 'b', 2, 'c', 3, 'd', 4]
```

2. L'idée est de parcourir la liste et calculer la tranche de chaque élément :

```
def compter(L: list)->list:
    """compter les entiers dans [10k; 10(k+1)[, 0<=k<=9"""
    s = [0]*10  # dix compteurs à 0
    for i in L:
        k = i // 10  # calcul de la tranche de i
        s[k] += 1  # ajouter 1 au compteur correspondant
```

```
    return s
>>> L = [rd.randint(0, 99) for _ in range(10)]
>>> L
[49, 97, 53, 5, 33, 65, 62, 51, 38, 61]
>>> rep = compter(L)
>>> rep
[1, 0, 0, 2, 1, 2, 3, 0, 0, 1]
```

3. Il suffit de parcourir la liste, de sommer les différents termes, et de diviser par le nombre d'éléments :

■■ Moyenne d'une liste de nombres

```
def moy(L:list)->float:
    """Calcule la moyenne d'une liste de nombres"""
    n = len(L)
    somme = 0  #  initialisation de la somme des éléments de \
    ↪ la liste
    for el in L:
        somme += el
    return somme / n
```

4. **■■ Variance d'une liste de nombres**

```
def var(L: list)->float:
    n = len(L)
    s = 0
    m = moy(L)
    for el in L:
        s += (el-m)**2
    return s / n
```