Масиви

Python, як сучасна об'єктно-орієнтована мова програмування має розширенні можливості роботи з послідовностями даних. Однак необхідно розділяти можливості стантарту Python та можливості стандартних бібліотек розштрення.

Python в мінімальній конфігурації (as is) включає два типи послідовностей: списки (list) та кортежі (tuple). В інтерактивній оболонці можна створити змінні кожного з цих типів.

```
>>> alist=[]
>>> blist=[1, 5, 6, 7, 9]
>>> type(alist)
<type 'list'>
>>> type(blist)
<type 'list'>
>>> atuple=()
>>> btuple=(7, 12, 4, 6,7)
>>> type(atuple)
<type 'tuple'>
>>> type(btuple)
<type 'tuple'>
```

Порівняно зі звичним розумінням масивів (як наприклад в pascal або basic) об'єкти типу **list** та **tuple** відрізняються декількома особливостями:

1. Об'єкти **list** та **tuple** можуть включати різнорідні за типом дані.

2. Об'єкти **list** можуть мати змінну довжину, а кожен елемент може змінювати своє значення.

3. Оскільки **list** та **tuple** є об'єктами, з ними асоційовано набір методів, що спрощує роботу.

Нижче наведено перелік основних методів об'єкту типу **list**. Замість слова list в синтаксичній схемі методу необхідно підставляти ідентифікатор екземпляру списка.

---

list.**append(*x*)** - додає в кінець списку новий елемент з значенням рівним ***x***:

```
>>> alist.append(3)
>>> alist
[3]
>>> blist.append("today")
>>> blist
[1, 5, 6, 7, 9, 'today']
```

---

list.**extend(*L*)** приєднує до списку інший елементи іншого списку ***(L)***

```
>>> alist.extend(blist)
>>> alist
[3, 1, 5, 6, 7, 9, 'today']
```

---

list.**insert(*i, x*)** вставляє в позицію ***i*** списку елемент ***x***. Нумерація елементів ведеться від 0. Якщо індекс ***i*** задано більшим ніж поточна довжина списку, елемент додається в кінець списку як командою **append**

```
>>> alist.insert(3, "paper")
>>> alist
[3, 1, 5, 'paper', 6, 7, 9, 'today']
```

List.remove(x) видаляє зі списку перший знайдений елемент зі значенням x. Якщо елемента з вказаним значенням у списку немає - генерується помилка.

```
>>> alist.remove(5)
>>> alist
[3, 1, 'paper', 6, 7, 9, 'today']
```

```
>>> alist
[3, 1, 6, 7, 9, 'today']
```

list.**pop(*[i]*)** видаляє зі списку і повертає елемент з вказаним індексом. Якщо індекс не вказаний, видаляється останній елемент. Якщо вказано індекс, що перевищує довжину списку - генерується помилка. Квадратні дужки вказують на необов'язковість параметра.

```
>>> alist.pop()
'today'
>>> alist
[3, 1, 6, 7, 9]
>>> g=alist.pop(2)
>>> g
6
>>> alist
[3, 1, 7, 9]
```

list.**index(*x*)** повертає індекс у списку першого елемента зі значенням ***x***. Якщо значення в списку відсутнє, генерується помилка.

```
>>> alist.index(7)
2
```

list.**count(*x*)** повертає кількість входжень елемента зі значенням ***x*** в список

```
>>> alist.count(1)
1
>>> alist.count(12)
0
```

list.**sort(*params*)** впорядковує список, за правилом описаним параметрами ***params***. Просте сортування можна проводити без параметрів

```
>>> alist
[1, 3, 7, 9]
```

list.**reverse()** переписує список в зворотньому напрямку

```
>>> alist.reverse()
>>> alist
[9, 7, 3, 1]
```

Списки підтримують операцію зрізу. Зріз можна визначити синтаксичною схемою **alist[m:n]** - двома параметрами розділеними двокрапкою. Один або обидва параметри можуть бути пропущені, тоді підставляється значення за замовчуванням: для першого параметра m=0 для другого параметра n=len(alist) - довжині списку. Операція зрізу виділяє елементи списку починаючи з m-го і закінчуючи (n-1)-м. Результат може бути присвоєно іншій змінній.

```
>>> blist=range(12)
>>> blist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> blist[0:5]
[0, 1, 2, 3, 4]
>>> blist[3:7]
[3, 4, 5, 6]
>>> blist[5:-3]
[5, 6, 7, 8]
```

Для розширення можливостей роботи з послідовностями, а також реалізації стандартних для інших мов програмування одновимірних та багатовимірних масивів застосовують зовнішні бібліотеки. Найпоширенішою з таких бібліотек є стандартна бібліотека **numpy** (*http://www.numpy.org*). Для роботи бібліотека повинна бути встановлена відповідним чином. Встановлену бібліотеку завантажують оператором import:

```
import numpy
```

або

```
import numpy as np
```

При використанні другого формату створюється ідентифікатор для швидкого виклику компоненів бібліотеки. Подальші приклади наведено для випадку першого формату.

Бібліотека **numpy** містить опис близько 20 додаткових типів даних, які можуть використовуватись для створення типізованих масивів.

Варіантів створення масивів існує декілька:

- перетворення стандартних об'єктів Python, таких як списки або котежі - метод доступнй для створення переважно одновимірних масивів;
- використання об'єктів - генераторів масивів - метод дозволяє створити масив довільної розмірності, заповнений відповідними елементами;
- читання компонентів масиву з дискового файлу - буде розглянуто пізніше;
- побудова масиву з рядка байтів;
- використання спеціальних бібліотечних функцій

Для початкового розгляду найбільш доречними є перші два варіанти.

Перетворення стандартних об'єктів

```
>>> alist=[12, 15, 18, 25]
>>> type(alist)
<type 'list'>
>>> a_arr=np.array(alist)
>>> a_arr
array([12, 15, 18, 25])
>>> type(a_arr)
<type 'numpy.ndarray'>
```

```
>>> btuple=(7, 8, 7, 8, 9)
>>> type(btuple)
<type 'tuple'>
>>> b_arr=np.array(btuple)
>>> b_arr
array([7, 8, 7, 8, 9])
>>> type(b_arr)
<type 'numpy.ndarray'>
```

```
>>> clist=[1, 3, 4, 5]
>>> dlist=[7, 2, 1, 3]
>>> elist=[8, 3, 3, 7]
>>> glist=[clist, dlist, elist]
>>> glist
[[1, 3, 4, 5], [7, 2, 1, 3], [8, 3, 3, 7]]
>>> gmatr=np.array(glist)
>>> gmatr
array([[1, 3, 4, 5],
       [7, 2, 1, 3],
       [8, 3, 3, 7]])
>>> type(glist)
<type 'list'>
>>> type(gmatr)
<type 'numpy.ndarray'>
```

Створювати масиви таким методом доцільно, якщо програмні компоненти, які їх формують і наповнюють значеннями уже створені та відлагоджені, і дані з них мають бути передані в масив, який створюється без змін.

Створення об'єктами-генераторами

```
>>> d_arr=np.zeros((2,3))
>>> d_arr
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

```
>>> e_arr=np.arange(8)
>>> e_arr
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
>>> f_arr=np.arange(2, 3, 0.1)
>>> f_arr
array([ 2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9])
```

```
>>> h_arr=np.ones((4,2))
>>> h_arr
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

```
>>> j_arr=np.linspace(1., 5., 7)
>>> j_arr
array([ 1.        ,  1.66666667,  2.33333333,  3.        ,  3.66666667,
        4.33333333,  5.        ])
```

Методи формування масиву **zeros** (нулі) та **ones** (одиниці) зручно використовувати для створення багатовимірних масивів, які пізніше будуть заповнені значеннями. Методи **arange** та **linspace** - для створення масивів з рівномірно розподіленими впорядкованими значеннями.

Цикли

В програмі мовою Python використовуються цикли двох типів:

- **for** - якщо необхідно виконати дію наперед відому кількість разів або провести обхід послідовності;
- **while** - якщо умови входу і/та виходу з циклу обчислюються всередині тіла циклу.

В обох випадках рядок ініціалізації циклу закінчується знаком двокрапки (**:**) а наступний рядок і усе тіло цикла водиться починаючи зі стандартного відступу - одна позиція табуляції.

Синтаксис оператора for:

```
for ind in {перелік можливих значень}:
      {operator 1}
      {operator 2}
      ...
      {operator N}
```

Змінна **ind** послідовно приймає усі можливі значення вказані в переліку. Перелік значень може бути заданий об'єктом будь-якого типу, що ітерується. Такими об'єктами є списки, кортежі, рядки, тощо. Для генерування "звичного" індексу використовують генератор range(n), який генерує n значень, починаючи з нуля з кроком 1.

```
>>> range(5)
[0, 1, 2, 3, 4]
```

Приклади використання циклів **for**

```
>>> for ind in range(5):
      print ind, ind*ind, ind**3

0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
```

```
>>> alist
[12, 15, 18, 25]
>>> for i in range(4):
      print alist[i]

12
15
18
25
```

```
>>> clist
[1, 3, 4, 5]
>>> for num in clist:
      print num

1
3
4
5
```

```
amatr=np.zeros((2,3))
print amatr
for i in range(2):
    for j in range(3):
        amatr[i,j]=input("A[ "+str(i)+'  '+str(j)+' ]=')
print amatr
```

Цикл типу while практично не відрізняється за харакером застосування від такогож циклу в інших мовах програмування.

```
>>> da=5
>>> while da<8:
      print da
      da+=1

5
6
7
```