Съставни типове. Списъци и кортежи.

Филип Андонов

23 май 2022 г.

Съдържание

Съставни типове

В повечето езици най-честият съставен тип са масивите. Те се харектиризират със следните свойства:

- Наредена съвкупност от еднотипни елементи
- Подредени последователно в паметта
- Адресируеми пряко
- Статични по отношение на размера си

Съставни типове

Всички елементи - “клетки” на такава структура са променливи от един и същ тип и всички действия, които могат да бъдат извършени над съдържанието на един, могат да се извършват и над всеки друг елемент. Същевременно това е структура с пряк достъп и всеки неин елемент се идентифицира с “името” си.

Съставни типове

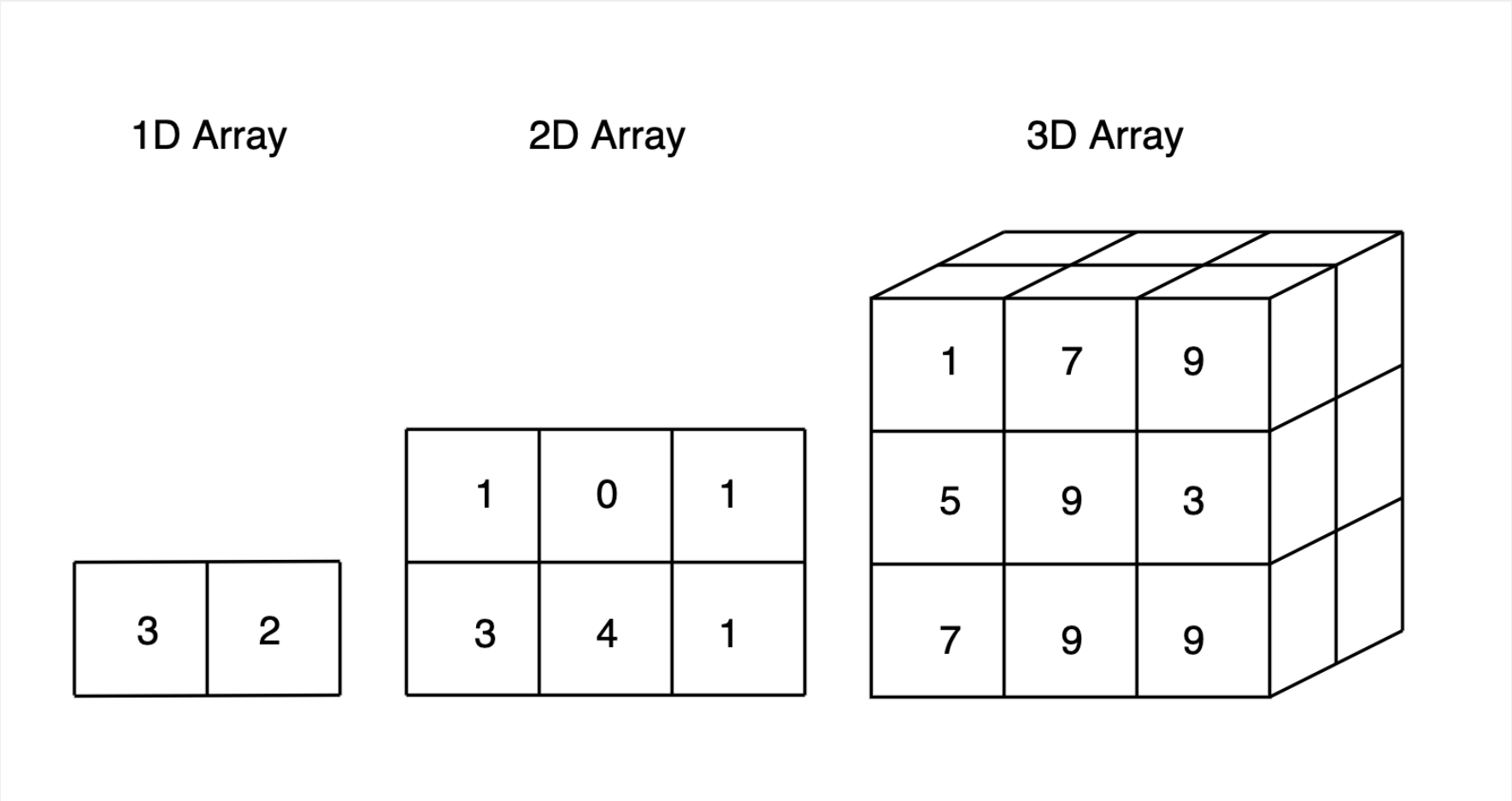
Декларирането на масив став като се укаже брой елементи и тип. При компилация се резервират последователни места от паметта според указаните брой и тип. Името на масива е всъщност име на указател от същия тип със стойност, равна на адреса на първия елемент.

Масивът не може да променя размера си. Размерът на масива трябва да е известен когато програмата се компилира, т.е. по време на проектиране, и не може да се задава по време на изпълнение.

Опитите за достъп извън декларираните граници на масив е често срещана грешка на програмиста, която не се разпознава при компилация, защото отместванията, чрез които се адресираме в масива са по принцип променливи, които получават стойност едва при изпълнение.

Правилото за идентифициране на елемент от масив в повечето езици за програмиране е заимствано от линейната алгебра, където "номерирането" на елементите на обекти от типа на вектори и на матрици става с посочване на позицията им долу вдясно (напр. A_{ji}) и се нарича индекс. По подобна аналогия с линейната алгебра, където обектите са разположени в пространства, и масивите имат "мерност".

Мерността и индексацията дават точно правило за това как да бъде идентифициран един конкретен елемент на масива. Всеки елемент-клетка на един n-мерен масив се идентифицира с наредена n-орка от индекси. Елементът има толкова индекси, колкото е "размерността" на пространството. Долу са илюстрирани "конгломерати" от еднотипни кутии, организирани като променливи-масиви:


1D Array
2D Array
3D Array
3
2
1
0
1
3
4
1
1
7
9
5
9
3
7
9
9

Нека на горните илюстрации позициите по всяко направление са “номерирани” от 1 до ... броя на елементите по това направление. Индексите по всяко направление се разглеждат като отделни множества, наричани “индексни множества”. Например, един тримерен масив има три индексни множества – по едно за всяко от направленията. Броят на елементите на структура като илюстрираните горе се пресмята като се умножат мощностите на индексните множества.

Съставни типове

От математическа гледна точка

- Едномерните масиви могат да се разглеждат като вектори
- Двумерните масиви могат да се разглеждат като матрици
- Тримерните масиви могат да се разглеждат като матрици на състояние

Масивите съхраняват линейни последователности от данни. Често трябва да съхраняваме последователности от данни (най-често числа) с двумерно разположение. В много езици се използват два индекса, за да декларираме двумерен масив.

Съставни типове

Python има 6 вградени типа за редици. Двете най-разпространени са списъците и кортежите. И двете са редици от стойности, които се адресират с индекс и на всяка позиция има елемент. Както и в C подобните езици първият елемент е с индекс 0. Стойностите на елементите в списъците и кортежите могат да са различни. Основната разлика между тях е, че списъците могат да се променят (mutable), докато кортежите не могат (immutable).

Съставни типове

Някои операции са приложими върху всички видове редици. Тези операции включват:

- Индексиране (indexing)
- Отрязъци (slicing)
- Добавяне
- Умножение (Multiplying)
- Проверка за принадлежност
- Функция за връщане на дължината на наредена редица
- Функции за намиране на минимална и максимална стойност

Индексиране

Спецификатори на конвертирането: Всички елементи в редици са номерирани – от 0 нагоре. Можем да се обръщаме към тях поотделно, както е показано по-долу.

Listing 1: Индексиране на списъци

```
1 >>> greeting = "Hello"
2 >>> greeting[0]
3 'H'
```

Индексиране

Това е индексиране – използваме индекс за да извлечем елемент. Всички редици могат да бъдат индексирани по този начин.
Когато използвате отрицателен индекс, Python брои отдясно наляво, т.е. от последния елемент. Последният елемент е на позиция -1 (а не -0, тъй като тогава ще е на същия индекс както и +0, т.е. първия)

Listing 2: Отрицателни индекси

```
1 >>> greeting = "Hello"
2 >>> greeting[-1]
3 'o'
```

Индексиране

Литералите на символните низове могат да се индексират директно, без да се използва променлива. Ако функция връща редица, можете да я индексирате директно.

```
>>> 'Hello'[1]
'e'
>>> fourth=raw_input('Year:')[3]
Year:2013
>>> fourth
'3'
```

Отрязъци

Както с индексирането можете да се обръщате и да боравите с отделни елементи, можете да използвате отрязване (slicing) за да достъпвате област от елементи. Това се прави чрез подаването на два индекса, разделени с двоеточие

```
1 >>> tag = '<a href="http://www.python.org">Python
      web site</a>'
2 >>> tag[9:30]
3 'http://www.python.org'
```

Отрязъци

Първият индекс е номерът на първия елемент, който искате да включите. Вторият индекс е номерът на първия елемент СЛЕД вашия отрязък. Обобщено: първият индекс включително, вторият индекс изключително

```
1 >>> tag = '<a href="http://www.python.org">Python
      web site</a>'
2 >>> tag[32:-4]
3 'Python web site'
```

Отрязъци

Вторият индекс сочи към елемент с индекс 10, който е 11-и и всъщност не съществува, но тъй като това е следващият след този, който искаме, това работи.

```
1 >>>numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 >>> numbers[7:10]
3 [8, 9, 10]
```

Отрязъци

След като можем да броим от края и можем да правим отрязъци, очакваме да можем да правим отрязъци от края. Това наистина е възможно, но се използва малка хитрост. Тя е да оставим втория индекс празен. Това указва на Python, че ние искаме да вземем „до края“, т.е. последните три, което именно беше и целта ни.

Listing 3: Отрязъци отзад напред

```
>>>numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[-3:-1]
[8, 9]
>>> numbers[-3:0]
[]
>>> numbers[-3:]
[8, 9, 10]
```

Отрязъци

По същия начин можем да броим и от началото. Вместо да указваме от къде и до къде да бъде отрязъка, ние можем просто кажем „първите три“ като оставим първия индекс празен.

Listing 4: Празни индекси в отрязъци

```
1 >>>numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 >>> numbers[:3]
3 [1, 2, 3]
```

Отрязъци

Ако искаме да копираме списък, можем да пропуснем и двата индекса. Така на практика казваме „от началото до края на списъка“. Това връща нов списък, съдържащ всички елементи от началото до края на оригиналния, тоест връща цялата редица.

Listing 5: Трик за копиране на списък

```
>>>numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Отрязъци

Параметрите, които давахме до този момент бяха за начало и край. Но всъщност отрязъците поддържат три параметъра – начало, край и стъпка.

Listing 6: Стъпки в отрязъци

```
>>> numbers[0:10:1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[0:10:2]
[1, 3, 5, 7, 9]
numbers[3:6:3]
[4]
>>> numbers[::4]
[1, 5, 9]
```

Стъпката не може да бъде 0, но може да бъде отрицателна, което означава извличане на елементите от дясно на ляво.

Listing 7: Отрицателна стъпка в отрязъци

```
>>> numbers[8:3:-1]
[9, 8, 7, 6, 5]
>>> numbers[10:0:-2]
[10, 8, 6, 4, 2]
```

Когато индексите са неявни, то Python прави правилното нещо – за положителна стъпка се движи от началото към края, а за отрицателна стъпка – от края към началото.

Добавяне

Редиците могат да се конкатенират като символните низове (всъщност низовете са редици и се конкатенират както и другите последователности в Python, но в много езици тази операция е допустима само за низове).

Listing 8: Добавяне на редици

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> 'Hello, ' + 'world!'
'Hello, world!'
>>> [1, 2, 3] + 'world!'
Traceback (innermost last):
File "<pyshell#2>", line 1, in ?
[1, 2, 3] + 'world!'
TypeError: can only concatenate list (not "string
    ") to list
```

Умножение на редици

Умножението на редица с число x създава нова редица, в която оригиналната редица е повторена x пъти.

Listing 9: Умножение на редици

```
>>> 'python' * 5
'pythonpythonpythonpythonpython'
>>> [42] * 10
[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

None

Празният списък се пише просто така: []. Но ако искаме да имаме празен списък с десет позиции? Можем да го инициализираме с нули [0]*10, но тогава той няма да е празен, а ще съдържа 10 нули. За да направим позиция, която не съдържа нищо, ползваме ключовата дума None.

Listing 10: Празна стойност

```
>>> sequence = [None] * 10
>>> sequence
[None, None, None, None, None, None, None, None,
    None, None]
```

Принадлежност

За да проверим дали дадена стойност се намира в дадена редица, ползваме булевия оператор in. Това работи за отделни елементи, но при символните низове можем да търсим на практика за подниз в низ.

Listing 11: Оператор in

```
>>> permissions = 'rw'
>>> 'w' in permissions
True
>>> 'x' in permissions
False
>>> users = ['Ivan', 'Petkan', 'Dragan']
>>> students = ['Ivan', 'Petkan', 'Dragan']
>>> 'Stoyan' in users
False
>>> 'Ivan' in users
True
>>> mailSubject = 'buy cheap viagra now!!!'
>>> 'viagra' in mailSubject
True
```

Дължина, мин/макс стойност

Всяка редица има дължина, т.е. текущ брой на елементите. Тя се намира с функцията len, на която се подава като параметър редицата, чиято дължина искаме да намерим. Аналогично работят функциите min и max, които намират съответно минималната и максималната стойност в редицата. Подредбата е за числовото представяне на стойностите.

Дължина, мин/макс стойност

Listing 12: Агрегатни функции в редици

```
numbers = [100, 34, 678]
>>>len(numbers)
3
>>>max(numbers)
678
>>>min(numbers)
34
>>>max(2, 3)
3
>>>min(9, 3, 2, 5)
2
```

Сортиране

Функцията sorted приема като параметри:

- редица или итератор, който да сортира
- key - функция, която ще се изпълни за всеки елемент и връщаща ключа за сортирането
- reverse – булева стойност определяща посоката на сортиране

Списъците имат метод sort(), който върши точно същото, но работи на място.

Listing 13: Метод sort

```
>>> c =['Italy','Bulgaria','Spain','Germany','
    France']
>>> b = sorted(c)
>>> print(b)
['Bulgaria','France','Germany','Italy','Spain']
>>> c.sort(key = len, reverse = True)
>>> c
['Bulgaria','Germany','France','Italy','Spain']
```

Основни операции върху списък

Можем да прилагаме стандартните операции на редиците върху списъци, но в допълнение можем да прилагаме и модифициращи операции. Това са операциите присвояване на елемент, изтриване на елемент, присвояване на отрязък и други.

Просто конвертиране

Присвояването се извършва, като се укаже в квадратни скоби позицията,на която искаме да запишем стойността, и отдясно на оператора за присвояване напишем самата стойност.

Listing 14: Присвояване в списъци

```
>>> students = ['Ivan', 'Petkan', 'Dragan']
>>> students[0] = 'Stoyan'
>>> students
['Stoyan', 'Petkan', 'Dragan']
```

Просто конвертиране

Изтриването става с оператор del, като синтаксисът е del име _ на _ редица [индекс], което изтрива от редицата елемента с дадения индекс. Например, ако е първият ден от отпуската ни и заглавията на писмата в пощата ни са тези в списъка по-долу можем да използваме командата del, за да се отървем от нежеланата поща. Сложността на функцията е O(n - i).

Listing 15: Изтриване от списък с del

```
1 >>> subjects = ['summer vacation offer','your
    salary for september','report for new students
    ASAP']
2 >>> del subjects[2]
3 >>> subjects
4 ['summer vacation offer', 'your salary for
    september']
```

Присвояване на отрязъци

Освен на отделни елементи, можем да присвояваме стойности на цели отрязъци. Може дори новата стойност да не е със същия брой елементи.

Listing 16: Присвояване на отрязъци

```
>>> name = list('Sly Stallone')
>>> name
['S', 'l', 'y', ' ', 'S', 't', 'a', 'l', 'l', 'o'
    , 'n', 'e']
>>> name[4:] = list('Schwarzenegger')
>>> name
['S', 'l', 'y', ' ', 'S', 'c', 'h', 'w', 'a', 'r'
    , 'z', 'e', 'n',
'e', 'g', 'g', 'e', 'r']
>>> name[0:3] = list('Arnold')
>>> name
['A', 'r', 'n', 'o', 'l', 'd', ' ', 'S', 'c', 'h'
    , 'w', 'a', 'r',
'z', 'e', 'n', 'e', 'g', 'g', 'e', 'r']
```

Копиране

За списъци операторът за присвояване работи като просто прави нова референция към същия списък.

Listing 17: Референции към списъци

```
1 >>>a = [1,2,3]
2 >>>b = a
3 >>>a.append(4)
4 >>>b
5 >>>[1, 2, 3, 4]
```

Затова ако искаме да копираме списък, използваме:

```
1 b = a[:]
```

Методи

В Python всички типове са класове. Това включва и контейнерните типове, които разглеждаме в тази глава. Така освен функции, които могат да се изпълняват върху тях, редиците имат методи, които се извикват с познатия от други езици синтаксис обект.метод (параметри).

Добавя един елемент в края на списък.

Listing 18: Метод append

```
1 >>> chores = ['clean the blackboard','open the
      windows','throw the garbage']
2 >>> chores.append('close the door')
3 >>> chores
4 ['clean the blackboard', 'open the windows', '
      throw the garbage', 'close the door']
```

Методът extend позволява добавянето на няколко стойности наведнъж към списък, които са представени като редица.

Listing 19: Метод extend

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> a.extend(b)
>>> a
[1,2, 3, 4, 5, 6]
```

На пръв поглед изглежда, че extend прави абсолютно същото като конкатенацията, но всъщност това не е така. Методът extend променя списъка, за който е извикан. Той е операция на място.

Listing 20: Метод extend срещу конкатенация

```
1 >>> a = [1,2,3]
2 >>> b = [4,5,6]
3 >>> a + b
4 [1, 2, 3, 4, 5, 6]
5 >>> a
6 [1, 2, 3]
```

index

Връща първия индекс, на който се намира стойността, подадена му като аргумент.

Listing 21: Метод index

```
1 >>> innovationRating = ['Sweden', 'Germany', '
    Denmark', 'Finland', 'Netherlands', '
    Luxembourg', 'Belgium', 'UK', 'Austria', '
    Ireland','France', 'Slovenia', 'Cyprus', '
    Estonia', 'Italy', 'Spain', 'Portugal', 'Chech
    _republic', 'Greece', 'Slovakia', 'Hungary', '
    Malta', 'Litva', 'Poland', 'Latvia', 'Romania'
    , 'Bulgaria']
2 >>> print("Bulgaria is on ",innovationRating.
    index('Bulgaria'))
3 +1,'place from ',len(innovationRating)
4 Bulgaria is on 27 place from 27
```

insert

Методът insert се използва за вмъкване на елемент в списък. Получава два аргумента – позиция, след която да бъде вмъкнат елемент и стойността на вмъкнатия елемент.

Listing 22: Метод insert

```
>>> numbers = [1,2,3,5,6,7]
>>> numbers.insert(3,'four')
>>> numbers
[1, 2, 3, 'four', 5, 6, 7]
```

pop

Методът pop премахва последния елемент от списък и го връща. С негова помощ можем да работим със списък като със стек.

Listing 23: Метод pop

```
>>> innovationRating.pop()
'Bulgaria'
>>> innovationRating
['Sweden', 'Germany', 'Denmark', 'Finland',
'Netherlands', 'Luxembourg', 'Belgium', 'UK',
'Austria', 'Ireland', 'France', 'Slovenia',
'Cyprus', 'Estonia', 'Italy', 'Spain', 'Portugal'
    ,
'Chech_republic', 'Greece', 'Slovakia', 'Hungary'
    ,
'Malta', 'Litva', 'Poland', 'Latvia', 'Romania']
```

Методът remove се използва за премахване на първото срещане на стойност в списъка. Сложността е $O(n)$.

Listing 24: Метод remove

```
>>> h = ['two','bee','or','not','two','bee']
>>> h
['two', 'bee', 'or', 'not', 'two', 'bee']
>>> h.remove('bee')
>>> h
['two', 'or', 'not', 'two', 'bee']
```

reverse

Методът reverse обръща редицата на елементите в списъка (и последните ще станат първи).

Listing 25: Метод reverse

```
>>> innovationRating
['Romania', 'Latvia', 'Poland', 'Litva', 'Malta',
'Hungary', 'Slovakia', 'Greece', 'Chech_republic',
'Portugal', 'Spain', 'Italy', 'Estonia', 'Cyprus',
'Slovenia', 'France', 'Ireland', 'Austria', 'UK',
'Belgium', 'Luxembourg', 'Netherlands', 'Finland',
'Denmark', 'Germany', 'Sweden', 'Bulgaria']
>>> innovationRating.reverse()
>>> innovationRating
['Bulgaria', 'Sweden', 'Germany', 'Denmark',
'Finland', 'Netherlands', 'Luxembourg', 'Belgium',
'UK', 'Austria', 'Ireland', 'France', 'Slovenia',
'Cyprus', 'Estonia', 'Italy', 'Spain', 'Portugal',
'Chech_republic', 'Greece', 'Slovakia', 'Hungary',
'Malta', 'Litva', 'Poland', 'Latvia', 'Romania']
```

sort

Методът sort сортира на място, т.е. не връща нов списък, а сортира списъка, за когото е извикан. Поведението му е аналогично на sorted.

Функции any и all

Функциите any и all са подобни на логически ИЛИ и И за редици.

any

Връща стойност True когато поне един от елементите се оценява на True.

all

Връща True само когато всичките елементи се оценяват на True.

Функции any и all

Таблица: Таблица на истинност на any и all

Оператор	any	all
Всички стойности са True	True	True
Всички стойности са False	False	False
Една стойност True, всички останали False	True	False
Една стойност False, всички останали True	True	False
Празна редици	False	True

Функции any и all

Listing 26: all и any

```
>>> a = [1,2,3,4]
>>> any(a)
True
>>> all(a)
True
>>> a.append(None)
>>> all(a)
False
>>> any(a)
True
```

Производителност

Списъците съдържат указатели към обекти, не самите обекти. Размерът на списъка в паметта зависи от броя на обектите, не от техния размер. Времето за достъп до отделен елемент е константа O(1) и не зависи от размера на списъка. Времето за добавяне на елемент в края на списъка е „амортизирана константа“ - когато трябва да се задели повече памет, се заделя повече от необходимото, за да се предотврати заделянето на всяко извикване (това предполага бърз алокатор). Ако искаме да използваме списъци в случаи, в които спецификата на имплементацията им ще доведе до загуба на производителност, можем да използваме blist от едноименния пакет.

Blist

Blist използва гъвкава хибридна структура, която комбинира масив и дърво и при преместване на елементи използва $O(logn)$ време. За да използваме blist, е необходимо да направим някои малки промени. Трябва разбира се да внесем модула, както и да създаваме списъци от този тип с функцията *blist*.

Listing 27: Използване на blist

```
1 items = [5, 6, 2]
2 from blist import blist
3 bitems = blist(items)
4 for element in bitems:
5     print(element)
```

Предимствата на blist можем да видим в таблицата. В нея n означава дължината на списъка, а k дължината на отрязъка.

Таблица: Производителност на blist и list

Употреба	blist	list
Вмъкване или премахване на елемент	O (log n)	O (n)
Правене на отрязъци	O (log n)	O (n)
Плитки копия (виж ??)	O (1)	O (n)
Промяна на отрязъци	O (log n + k)	log O (n + k)

Кортежи

Кортежите са редици, също като списъците. Разликата е, че кортежите не могат да се променят. Символните низове също не могат да се променят. Разделянето със запетаи на стойности, които не са оградени с никакви скоби, автоматично ги превръща в кортеж. Същото може да се направи и ако се изброените стойности се поставят в кръгли скоби (ред 3). Неизменимостта понякога води до объркването, че на променлива от такъв тип не можем да присвоим нова стойност. Това разбира се не е така. Ние не можем да променяме съдържанието частично, т.е. всяка промяна води до създаването на нова стойност.

Празният кортеж се пише с кръгли скоби, в които няма нищо (ред 4). Ако искаме да направим кортеж от единична стойност, трябва да добавим запетая (ред 6), въпреки че стойността е една. По този начин отличаваме литерал от кортеж.

Listing 28: Създаване на кортежи

```
1 >>> 1,2,3
2 (1, 2, 3)
3 >>> (1,2,3)
4 >>> ()
5 ()
6 >>> 42,
7 (42,)
```

Функцията tuple работи по същия начин като функцията list – тя приема редица като аргумент и я конвертира в кортеж. Ако аргументът вече е кортеж, то той се връща непроменен:

Listing 29: Функцията tuple

```
>>> tuple([1, 2, 3])
>>> (1, 2, 3)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple((1, 2, 3))
(1, 2, 3)
```

Номерирани итерации

В някои случаи искаме да итерираме върху редица от обекти и в същото време да имаме достъп до индекса на текущия обект. Това е наложително, когато искаме да променяме редица, върху която итерираме. Причината е, че в конструкцията
for element in sequence element е променлива – копие и променяйки я ние променяме копие, а не стойността на текущия елемент на редицата. Функцията enumerate() добавя брояч към това, върху което итерираме. Тя има и незадължителен втори параметър, който указва коя да е първата стойност на брояча.

Listing 30: Номериране на итерации с enumerate

```
numlist = [1,2,3,4,5]
for num in numlist:
    num = num * 2
print(numlist)
for index, num in enumerate(numlist):
    numlist[index] = num * 2
print(numlist)
```

Нека искаме да заменим всеки символен низ, който съдържа подниза 'xxx', с [CENSORED] в списък със символни низове.

Listing 31: Промяна на текущия елемент с enumerate

```
1 txt = "Cheap␣xxx␣pictures␣and␣xxx␣videos!␣Quality
    ␣mature␣xxx␣content!␣Get␣your␣xxx␣pass␣now!"
2 text = txt.split('␣')
3 for index,string in enumerate(text):
4     if 'xxx' in string:
5         text[index] = '[CENSORED]'
6 txt = "␣".join(text)
7 print(txt)
```

Генериране на списъци

Можем да използваме цикъл, за да генерираме стойности в списък. Разбира се, това може да се направи със стандартен цикъл и използването на метод append, но Python предлага съкратен запис, в който не е необходимо да указваме операцията.
Синтаксисът е
[израз върху променлива for променлива in range() if условие]

Listing 32: Генериране на списъци

```
>>> [x * 2 for x in range(10,20)]
[20, 22,24, 26, 28, 30, 32, 34, 36, 38]
>>> [x * 2 for x in range(10,20) if x % 3 == 0]
[24, 30,36]
```

Понякога се налага да правим многомерни структури, например таблица/матрица. За целта в Python можем да направим списък от списък. Ако напишем A=[[]]*10 и след това го изведем на екрана с print ще изглежда, че сме създали списък от 10 елемента, които са списъци. Всъщност това е почти така, но всеки елемент е различна референция към един и същ списък. Всяка промяна в който и да е елемент на A се отразява във всички останали. За да създадем списък от 10 различни списъка, трябва да напишем:

Listing 33: Многомерни списъци

```
1 a = [[] for i in range(10)]
2 b = [[None] * 10 for i in range(10)]
3 b[1][2] = 8
```

Задачи

- Да се създаде програма, което проверява дали даден списък е сортиран. Например за списъка ([1,2,2]) програмата трябва да казва, че е сортиран, докато за (' b ',' a ') трябва да казва, че не е сортиран.
- Две думи са анаграми, ако можем да пренаредим буквите на едната, за да напишем другата. Да се създаде програма, която проверява дали два символни низа са анаграми.
  „Vladimir Nabokov“ = „Vivian Darkbloom“
  „rocket boys“ = „october sky“
- Да се създаде програма, която получава списък и проверява дали който и да е от елементите му се повтаря повече от веднъж.
- Да се създаде програма, която взима списък и прави нов, в който всеки от елементите на първия се среща точно веднъж.

Thanks for using Focus!

References