**Упорядоченный список**

Все это можно представить в виде упорядоченно списка, который в Python задается с помощью оператора квадратных скобок:

[элемент1, элемент2, …, элементN]

Например, для хранения городов можно задать такой список:



Здесь синей рамкой отмечен сам список, внутри которого располагаются элементы. Если мы посмотрим тип объекта, на который ссылается переменая lst:

type(lst)

то увидим значение «list». Это как раз и есть тип списка. То есть, через переменную lst мы можем работать со списком в целом, и первое, что нас здесь интересует: как обратиться к определенному элементу этого списка?

Для этого используется такой синтаксис: список[индекс]

Например,

lst[0]

lst[2]

Но, если мы укажем не существующий индекс:

lst[5]

то возникнет ошибка. Чтобы этого избежать нам надо знать значение последнего индекса.

Для этого можно воспользоваться функцией len(список)

которая возвращает число элементов в списке:

len(lst)

вернет значение 4. Но, так как индексы начинаются с нуля, то последний индекс будет равен:

lastIndex = len(lst) – 1

То есть, можно записать вот так:

lst[len(lst)-1]

но можно и проще, вот так:

lst[-1]

Этот пример показывает, что при отрицательных индексах, мы начинаем движение с конца списка и значение -1 дает самый последний элемент.

Далее, для перебора элементов списка в Python очень удобно использовать цикл for:

lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]

**for** city **in** lst:

**print**(city)

Смотрите, как это легко и просто делается! Конечно, мы можем распечатать весь контейнер целиком, просто записав:

**print**(lst)

Но здесь нет перебора всех элементов, а просто печать содержимого.

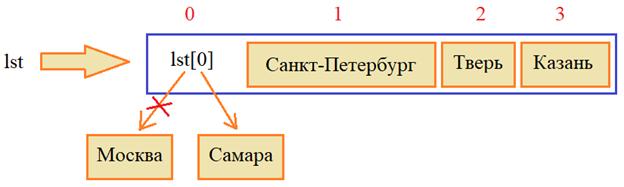
Но вот список list относится к изменяемым типам, то есть, мы можем изменить его состояние, не создавая нового объекта. В самом простом случае мы можем воспользоваться таким синтаксисом:

список[индекс] = значение

Например, так:

lst[0] = "Самара"

Теперь первый элемент не «Москва», а «Самара». В действительности, здесь произошло следующее:



Мы поменяли значение ссылки lst[0] первого элемента списка. Изначально она ссылалась на объект «Москва», а затем, стала ссылаться на объект «Самара». Прежний объект автоматически удаляется сборщиком мусора. Вот так происходит изменение элементов списка, то есть, меняются значения ссылок на новые объекты.

Теперь вернемся к вопросу изменения элементов списка внутри цикла for. Предположим, у нас имеется список чисел:

digs = [-1, 0, 5, 3, 2]

и в цикле мы хотим его изменить на их квадраты. Для этого запишем цикл в таком виде:

digs = [-1, 0, 5, 3, 2]

**for** x **in** range(5):

    digs[x] \*\*= 2   *#digs[x] = digs[x]\*\*2*

**print**(digs)

Или, чтобы не указывать конкретное число в функции range, ее можно переписать так:

**for** x **in** range(len(digs)):

И программа будет работать со списком произвольной длины.

Во всех наших примерах мы создавали списки небольшой длины и потому могли их запросто записать в программе. Но что если требуется создать список размерностью в 100 или 1000 элементов? Для этого можно воспользоваться такой конструкцией, например:

A = [0]\*1000

создает список из 1000 элементов со значением 0. Фактически, мы здесь сначала создали список из одного нулевого элемента, а затем, размножили его до тысячи. Или, можно сделать так:

A = ["none"]\*100

Получим 100 элементов со строкой «none». И так далее. Кстати, если требуется создать пустой список, то это будет так:

Для наглядности, предположим, пользователь вводит N чисел с клавиатуры (N<100) и пока он вводит положительные значения, мы их добавляем в список. Как только он ввел какое-либо отрицательное число, считывание прекращается и мы вычисляем среднее арифметическое введенных значений. Это можно реализовать так:

digs = [0]\*100

N = 0; x = 0

**while** x >= 0:

   x = int(input("Введите целое число: "))

   digs[N] = x

   N += 1

S = 0

**for** x **in** range(N):

    S += digs[x]

S = S/N;

**print**("S = %f, N = %d"%(S, N))

Теперь, когда мы в целом познакомились со списками, отметим следующие моменты. Список может состоять из произвольных данных, например:

t = ["строка", 5, 5.7, True, [1,2,3]]

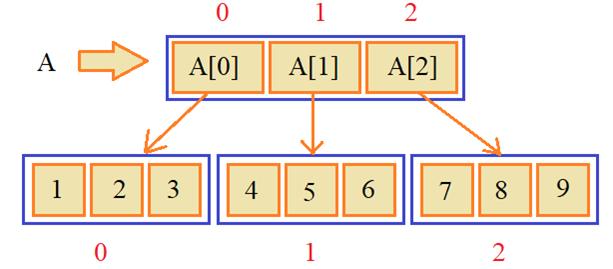
Причем, его длина

len(t)

будет равна 5, т.к. последний элемент – вложенный список здесь воспринимается как один отдельный элемент.

И этот пример показывает как можно создавать двумерные списки:

A = [[1,2,3], [4,5,6], [7,8,9]]



Для доступа к конкретному числу следует сначала обратиться к первому списку:

A[1]

а, затем, ко второму:

A[1][0]

Получим значение 4. С этими списками можно выполнять все те же самые операции, о которых мы говорили ранее, например, изменить значение:

A[1][2] = -1

Далее, списки можно объединять друг с другом, используя оператор +:

[1,2,3] + ["Москва", "Тверь"]

Используя этот оператор, можно добавлять новые элементы к списку:

digs = [1,2,3,4]

digs = digs + [5]

digs += [6]

или в начало:

digs = ["числа"]+digs

И здесь обратите внимание, что мы объединяем именно списки, то есть, вот такая запись:

digs = digs+3

приведет к ошибке, т.к. 3 – это число, а не список.

Следующий оператор:

3 **in** digs

возвращает True, если элемент, записанный слева, присутствует в списке, указанный справа. Иначе, значение False:

-1 **in** digs

Или, можно делать так:

[1,2,3] **in** A

То есть, в качестве элемента может быть любой тип данных.

**Следующие две полезные функции:**

digs = [1,2,3,4]

max(digs)

min(digs)

находят минимальное или максимальное числовое значение. И если в списке имеется не числовой элемент:

digs += "a"

то эти функции приводят к ошибкам.

Также можно вычислять сумму элементов числового списка:

d = [1,2,3,4]

sum(d)

выполнять сортировку чисел по возрастанию:

d = [-1, 0, 5, 3, 2, 5]

sorted(d)

или, по убыванию:

sorted(d, reverse=True)

Эта функция возвращает новый объект-список, **прежний d остается без изменений.**

**Наконец, можно сравнивать списки между собой:**

[1,2,3] == [1,2,3]

[1,2,3] != [1,2,3]

[1,2,3] > [1,2,3]

В последнем сравнении получим False, т.к. списки равны, но если записать так:

[10,2,3] > [1,2,3]

то первый список будет больше второго. Здесь сравнение больше, меньше выполняется по тому же принципу, что и у строк: перебираются последовательно элементы, и если текущий элемент первого списка больше соответствующего элемента второго списка, то первый список больше второго.

И аналогично, при сравнении меньше:

[10,2,3] < [1,2,3]

Все эти сравнения работают с однотипными данными:

[1,2, "abc"] > [1,2, "abc"]

сработает корректно, а вот так:

[1,2,3] > [1,2, "abc"]

Произойдет ошибка, т.к. число 3 не может быть сравнено со строкой «abc».

**Задания**

1. Дан список [-1, 0, 5, 3, 2].

Необходимо изменить его, увеличив каждое значение на 7.2.

2. Пользователь вводит с клавиатуры N значений (строки или числа).

На их основе сформировать список, состоящий из продублированных элементов.

(Например, из значений 1, 5, "abc" формируется список [1, 1, 5, 5, "abc", "abc"]).

3. Пользователь вводит N значений в список.

Необходимо проверить: было ли введено число 5.

**Списки: срезы и методы**

Для списков применим механизм срезов, о котором мы уже говорили, рассматривая строки. Например, пусть у нас имеется список из городов:

lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]

Используя синтаксис: список[start:end] можно выделять элементы, начиная с индекса start и заканчивая, но не включая индекс end. В частности, вот такая конструкция:

lst[1:3]

возвратит список из двух городов:

['Санкт-Петербург', 'Тверь']

То есть, здесь **создается новый объект list**, содержащий эти элементы:

lst2 = lst[2:4]

Прежний список lst не меняется.

Если индексы принимают отрицательные значение, то отсчет идет с конца списка:

lst[-2:-1]

lst[0:-1]

У срезов можно записывать любые числовые индексы к ошибкам это не приведет. Например:

lst[1:9999]

вернет список со 2-го элемента и по последний:

['Санкт-Петербург', 'Тверь', 'Казань']

Этот же результат можно получить и так:

lst[1:]

lst[:3]

Если не указывать ни начало, ни конец, то будет возвращен список:

lst[:]

Спрашивается: создает ли данная операция копию списка?

Да, создается и в этом легко убедиться, записав такие строчки:

c = lst[:]

**print**(id(c), id(lst))

И мы увидим разные значения id, которые говорят, что обе переменные ссылаются на разные списки.

Также копию списка, можно сделать с помощью функции-конструктора list:

c=list(lst)

Далее, в срезах можно указывать шаг следования (по умолчанию он равен 1). Для этого пишется еще одно двоеточие и указывается шаг:

lst[::2]

получим:

['Москва', 'Тверь']

Или, такие варианты:

lst[1:4:2]

lst[:4:3]

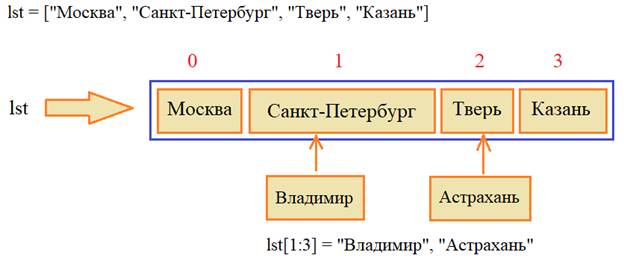
lst[1::2]

Если указать отрицательный шаг, то перебор будет происходить в обратном порядке:

lst[::-1]

Так как список – это изменяемый объект, то мы можем срезам присваивать новые значения. Делается это таким образом:

lst[1:3] = "Владимир", "Астрахань"



В результате, получаем список: ['Москва', 'Владимир', 'Астрахань', 'Казань']

Или даже так. Большему срезу присвоить меньшее число элементов:

lst[0:3] = 'Пермь', 'Пенза'

В итоге получаем список:

['Пермь', 'Пенза', 'Казань']

**если нам** нужно просто удалить какой-либо элемент, то это делается с помощью оператора del:

**del** lst[1]

В результате будет удален элемент с индексом 1 из списка lst:

['Пермь', 'Казань']

**Методы списков**

Давайте теперь предположим, что у нас имеется список из чисел:

a = [1, -54, 3, 23, 43, -45, 0]

и мы хотим в конец этого списка добавить значение. Это можно сделать с помощью метода:

**a.append(100)**

И обратите внимание: метод append **ничего не возвращает**, то есть, он меняет сам список благодаря тому, что он относится к изменяемому типу данных. Поэтому писать здесь конструкцию типа

a = a.append(100)

категорически не следует, так мы только потеряем весь наш список!

Причем, мы в методе append можем записать не только число, но и другой тип данных, например, строку:

a.append("hello")

тогда в конец списка будет добавлен этот элемент. Или, булевое  значение:

a.append(True)

Или еще один список:

a.append([1,2,3])

И так далее. Главное, чтобы было указано **одно конкретное значение**. Вот так работать не будет:

a.append(1,2)

Если нам нужно вставить элемент в произвольную позицию, то используется метод

**a.insert(3, -1000)**

Здесь мы указываем индекс вставляемого элемента и далее значение самого элемента.

Следующий **метод remove** удаляет элемент по значению:

a.remove(True)

a.remove('hello')

Он находит первый подходящий элемент и удаляет его, остальные не трогает. Если же указывается несуществующий элемент:

a.remove('hello2')

то возникает ошибка.

**Еще один метод для удаления**

a.pop()

выполняет удаление последнего элемента и при этом, **возвращает его** значение. В самом списке последний элемент пропадает. То есть, с помощью этого метода можно сохранять удаленный элемент в какой-либо переменной:

end = a.pop()

Также в этом методе можно указывать индекс удаляемого элемента, например:

a.pop(3)

**Если нам нужно очистить весь список** – удалить все элементы, то можно воспользоваться методом:

a.clear()

Получим пустой список.

**Следующий метод**

a = [1, -54, 3, 23, 43, -45, 0]

c = a.copy()

возвращает копию списка. Это эквивалентно конструкции:

c = list(a)

Следующий **метод count** позволяет найти число элементов с указанным значением:

c.count(1)

c.count(-45)

Если же нам нужен **индекс определенного значения,** то для этого используется метод index:

c.index(-45)

c.index(1)

возвратит 0, т.к. берется индекс только первого найденного элемента.

Но, мы здесь можем указать стартовое значение для поиска:

c.index(1, 1)

Здесь поиск будет начинаться с индекса 1, то есть, со второго элемента. Или, так:

c.index(23, 1, 5)

Ищем число 23 с 1-го индекса и по 5-й не включая его. Если элемент не находится

c.index(23, 1, 3)

то метод приводит к ошибке. Чтобы этого избежать в своих программах, можно вначале проверить: существует ли такой элемент в нашем срезе:

23 **in** c[1:3]

и при значении True далее уже определять индекс этого элемента.

**Следующий метод**

c.reverse()

меняет порядок следования элементов на обратный.

**Последний метод**, который мы рассмотрим, это

c.sort()

выполняет сортировку элементов списка по возрастанию. Для сортировки по убыванию, следует этот метод записать так:

c.sort(reverse=True)

Причем, этот метод работает и со строками:

lst = ["Москва", "Санкт-Петербург", "Тверь", "Казань"]

lst.sort()

|  |  |
| --- | --- |
| Метод | Описание |
| append() | Добавляет элемент в конец списка |
| insert() | Вставляет элемент в указанное место списка |
| remove() | Удаляет элемент по значению |
| pop() | Удаляет последний элемент, либо элемент с указанным индексом |
| clear() | Очищает список (удаляет все элементы) |
| copy() | Возвращает копию списка |
| count() | Возвращает число элементов с указанным значением |
| index() | Возвращает индекс первого найденного элемента |
| reverse() | Меняет порядок следования элементов на обратный |
| sort() | Сортирует элементы списка |

**Задания**

1. Пользователь вводит с клавиатуры числа, до тех пор, пока не введет число 0. На основе введенных данных нужно сформировать список, состоящий из квадратов введенных чисел.

2. Написать программу удаления из списка всех номеров с кодом «+7».

['+7912123456', '+7915213456', '+6915213456', '+4915213456']

3. Написать программу циклического сдвига элементов списка влево. Например, дан список: [1,2,3,4,5,6]

после сдвига на один элемент влево, должны получить: [2,3,4,5,6,1]

Реализовать через цикл, перебирая все элементы.

4. Написать аналогичную программу циклического сдвига, но теперь вправо.

# Инструмент list comprehensions

Списки в Python играют большую роль и, прежде чем идти дальше по материалу, рассмотрим некоторые упрощения при работе с ними. То, о чем мы сейчас будем говорить, в английской литературе называется **List comprehensions**

Представьте, что нам нужно создать список, состоящий из квадратов чисел, определенных от 0 до N-1. Используя текущие знания, это можно было бы реализовать так:

A=[]

N=10

**for** x **in** range(N):

     A.append(x\*\*2)

**print**(A)

А теперь этот же алгоритм с использованием list comprehensions:

N=10

A = [x\*\*2 **for** x **in** range(N)]

**print**(A)

Смотрите, все создание списка записано в одну строчку. И, кроме того, этот вариант будет работать быстрее предыдущей программы, т.к. Python оптимизирует работу таких конструкций.

Давайте теперь разберемся в этом синтаксисе. Вначале мы указываем, что будем делать с переменной x. Казалось бы, переменная нигде не задана, а мы уже говорим что с ней делать. Да, это так, это такой элемент синтаксиса list comprehensions. Далее, через пробел мы записываем цикл for и уже там указываем эту переменную x и говорим как она будет меняться. То есть, эта временная переменная x существует только внутри списка и пропадает после его создания.

Теперь немного усложним этот пример, и предположим, что нам нужно создать список только из четных чисел. Опять же, обычным способом мы могли бы это сделать так:

A=[]

N=10

**for** x **in** range(N):

**if** x%2 == 0:

          A.append(x\*\*2)

**print**(A)

А через list comprehensions это записывается следующим образом:

N=10

A = [x\*\*2 **for** x **in** range(N) **if** x%2 == 0]

**print**(A)

То есть, после цикла мы можем записать любое условие и если оно становится истинным, то формируется новый элемент списка. Видите, как это просто можно реализовать.

Разумеется, вместо возведения в степень мы можем прописывать любые функциональные выражения, или записать просто x:

A = [x **for** x **in** range(N) **if** x%2 == 0]

Тогда получим последовательность четных чисел.

Или же формулу вычисления значений линейной функции:

A = [0.5\*x+1 **for** x **in** range(N)]

И так далее, здесь может быть любое выражение. Также можно оперировать не только числами, но и другими типами данных, например, строками:

cities = ["Москва", "Тверь", "Рязань", "Ярославль", "Владимир"]

A = [city **for** city **in** cities **if** len(city) < 7]

**print**(A)

Здесь мы формируем список из названий городов, длины которых меньше семи символов. По этому принципу можно строить самые разные алгоритмы формирования списков.

# Алгоритмы обработки списков

## Разбивка целого положительного числа по цифрам

Программа на Python будет выглядеть следующим образом:

x = int(input("Введите целое положительное число: "))

digs = []

**while** x:

    digs.append(x%10)   *#берем последнюю цифру числа*

    x = x//10           *#отбрасываем последнюю цифру числа*

**print**(digs)

Чтобы цифры шли по порядку: слева-направо, мы их будем добавлять в начало списка:

digs = [x%10] + digs

## Программа, меняющая порядок следования элементов в списке

*# программа reverse*

N = 11

A = list(range(N))

**print**(A)

**for** i **in** range(N//2):

    A[i], A[N-i-1] = A[N-i-1], A[i]

**print**(A)

## Сортировка методом выбора

Идея этого метода сортировки довольно проста. Классический пример для объяснения подобных алгоритмов – выстроить людей по росту, допустим по возрастанию.

*# сортировка методом выбора*

A = [2, 2, -1, -5, 55, 34, 0, 10]

N = len(A)

**for** i **in** range(N-1):

**for** j **in** range(i+1, N):

**if** A[i] > A[j]:

            A[i], A[j] = A[j], A[i]

**print**(A)

Одной из наиболее важных особенностей списков

Одной из наиболее важных особенностей списков является создание псевдонимов при операции присваивания списка другой переменной. Псевдонимы — это переменные, которые имеют разные имена, но содержат одинаковые адреса памяти. Данная особенность важна и ее необходимо учитывать, т.к. можно случайно, работая с одной переменной, испортить значения, хранящиеся в другой. Рассмотрим эти моменты детальнее на следующем примере.

list1=[1,2,3,4,5]

print(list1) #[1, 2, 3, 4, 5]

list2=list1

print(list2) #[1, 2, 3, 4, 5]

list2[1]="Hello"

print(list2) #[1, 'Hello', 3, 4, 5]

print(list1) #[1, 'Hello', 3, 4, 5]

Вначале у нас есть список list1=[1, 2, 3, 4, 5]. Далее мы создаем новый список list2 и присваиваем ему список list1. После этой операции переменная list2 содержит тот же адрес в памяти, что и переменная list1, т.е. фактически ссылается на тот же самый список.

Поэтому, когда мы, используя переменную list2, изменили второй элемент списка на слово «Hello», то при выводе списка как через переменную list2, так и через переменную list1, мы видим обновлённый список. Если необходимо проверить, ссылаются ли две разные переменные на один и тот же список, то можно использовать следующий подход с помощью оператора is, который проверяет, являются ли две переменный одним и тем же объектом:

Если же нам необходимо скопировать элементы из существующего списка в новый (т.е. создать новый объект с такими же значениями), то можно использовать один из следующих способов:

■ использовать функцию copy();

■ использовать функцию-конструктор списка list();

■ использовать срез всего списка [:].