Модуль 4. Список как изменяемая последовательность

[Введение 1](#_Toc147787862)

[Списки - операторы и функции работы с ними 1](#_Toc147787863)

[Функции работы со списками 4](#_Toc147787864)

[Практика. Перебор списков 6](#_Toc147787865)

[Операторы списков 6](#_Toc147787866)

[Срезы списков и сравнение списков 8](#_Toc147787867)

[Сравнение списков 11](#_Toc147787868)

[Основные методы списков 13](#_Toc147787869)

[Практика методы списков 18](#_Toc147787870)

[Вложенные списки, многомерные списки 19](#_Toc147787871)

[Генераторы списков (List comprehensions) 23](#_Toc147787872)

[Условия в генераторах списков 25](#_Toc147787873)

[Вложенные генераторы списков 26](#_Toc147787874)

[Практика списки 30](#_Toc147787875)

[Комбинаторика. Модуль itertools 30](#_Toc147787876)

[Комбинаторика: сочетания 30](#_Toc147787877)

[Комбинаторика: размещения 30](#_Toc147787878)

[Комбинаторика: размещение с повторениями 30](#_Toc147787879)

[Практика комбинаторика 31](#_Toc147787880)

# Введение

Структуры данных

Структуры данных – это, по сути, и есть структуры, которые могут хранить некоторые данные вместе. Другими словами, они используются для хранения связанных данных.

В Python существуют четыре встроенных структуры данных: список, кортеж, словарь и множество. Посмотрим, как ими пользоваться, и как они могут облегчить нам жизнь

# Списки - операторы и функции работы с ними

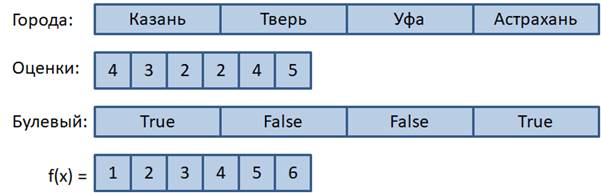
Список – это структура данных, которая содержит упорядоченный набор элементов, т.е. хранит последовательность элементов. Это легко представить, если вспомнить список покупок, в котором перечисляется, что нужно купить, с тем лишь исключением, что в списке покупок каждый элемент обычно размещается на отдельной строке, тогда как в Python они разделяются запятыми.

shoplist = ['яблоки', 'манго', 'морковь', 'бананы']

Задать список в программе на Python очень просто, ставятся квадратные скобки и внутри них через запятую перечисляются его элементы:

Как только список создан, можно добавлять, удалять или искать элементы в нём. Поскольку элементы можно добавлять и удалять, мы говорим, что список – это изменяемый тип данных, т.е. его можно модифицировать.

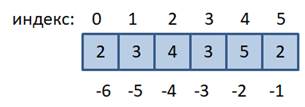
print('Я должен сделать', len(shoplist), 'покупки.')  
print('Покупки:', end=' ')  
for item in shoplist:  
 print(item, end=' ')  
print('\nТакже нужно купить риса.')  
shoplist.append('рис')  
print('Теперь мой список покупок таков:', shoplist)  
print('Отсортирую-ка я свой список')  
shoplist.sort()  
print('Отсортированный список покупок выглядит так:', shoplist)  
print('Первое, что мне нужно купить, это', shoplist[0])  
olditem = shoplist[0]  
del shoplist[0]  
print('Я купил', olditem)  
print('Теперь мой список покупок:', shoplist)



Чтобы оперировать списком через переменную, используется оператор присваивания:

marks = [2, 3, 4, 3, 5, 2]

Давайте теперь посмотрим, как эту конструкцию, этот список использовать в программировании? Например, вычислить средний балл по оценкам? Для этого нам надо уметь обращаться к отдельным элементам списка. Как это сделать? Так как список – это упорядоченная коллекция, то все его элементы имеют свой порядковый номер – индекс, начиная с нулевого и заканчивая последним.



Используя эти индексы и синтаксис:

список[индекс]

мы можем обращаться к отдельным элементам. Например:

marks[0]

marks[2]

И для вычисления среднего балла среди шести оценок, получим формулу:

avg = (marks[0] + marks[1] + marks[2] + marks[3] + marks[4] + marks[5]) / 6

Это простейший пример того, как можно хранить и использовать данные списка. Причем индексирование здесь работает так же, как и со строками. Если указать несуществующий индекс:

marks[10]

то получим ошибку. А чтобы обратиться к последнему элементу, можно использовать отрицательный индекс:

marks[-1]

То есть, мы здесь также имеем наборы отрицательных индексов, идущих от конца к началу списка.

Список – изменяемый тип данных

В отличие от строк, списки в Python относятся к изменяемым типам данных. То есть, мы можем изменить ранее хранимое значение. Например, студен пересдал первую двойку на тройку и мы хотим внести это изменение в список marks. Сделать это можно через оператор присваивания:

marks[0] = 3

Все, теперь первое значение равно 3. Как вы помните, со строками такая операция приводила к ошибке, так как строки – это неизменяемый тип. Но со списками мы так делать можем и в этом их кардинальное отличие от строк. Список – динамическая структура данных, который может меняться в процессе работы программы.

Мало того, списку, состоящему из чисел, мы легко можем присвоить любой другой тип данных, например, строку:

marks[1] = "удовл."

Вообще списки могут содержать самые разные типы данных, например:

lst = ["Москва", 1320, 5.8, True, "Тверь", False]

в том числе и другие, вложенные списки:

lst2 = [1, 2.5, [-1, -2, -3], 4]

О вложенных списках мы еще будем говорить.

Если нам нужно создать пустой список, то достаточно записать квадратные скобки без элементов:

a = []

или, воспользовавшись специальной функцией:

b = list()

которая создает новый пустой список. Если же ей в качестве аргумента указать другой список:

a = list([True, False])

то будет создан новый список с тем же самым содержимым. Также мы можем передать ей строку:

list("Python")

тогда получим список, состоящий из отдельных символов этой строки. Вообще на вход функции list() можно передавать любой перебираемый объект, на основе которого формируется новый список. Такие перебираемые объекты еще называются итерируемыми, но мы о них будем говорить далее.

## Функции работы со списками

Язык Python содержит несколько удобных встроенных функций для работы со списками:

* len() – определение числа элементов в списке (длина списка);
* max() – для нахождения максимального значения;
* min() – для нахождения минимального значения;
* sum() – для вычисления суммы;
* sorted() – для сортировки коллекции.

Для начала воспользуемся уже знакомой нам функцией len для определения длины списка:

len(marks)

Соответственно, для пустого списка:

len([])

она возвращает 0.

Две из них min() и max() нам уже знакомы. Сейчас мы увидим, как их можно применять к спискам. Сформируем список значений температуры по дням города Москвы:

t = [23.5, 25.6, 27.3, 26.0, 30.4, 29.5]

Теперь, чтобы найти максимальное и минимальное значения, достаточно вызвать функции:

max(t)

min(t)

Для подсчета суммы всех значений, запишем функцию sum:

sum(t)

А вычислить среднюю температуру можно следующим образом:

sum(t)/len(t)

Видите, как просто это делается на уровне функций.

Наконец, последняя функция sorted, если ее вызвать с одним аргументом:

sorted(t)

то она возвратит новый список с отсортированными значениями по неубыванию (или, как часто говорят, по возрастанию). И, обратите внимание, эта функция не меняет прежний список t, она именно возвращает новый список с отсортированными значениями. Очевидно, чтобы сохранить результат работы этой функции, следует использовать переменную, например, так:

t\_sort = sorted(t)

Если же нам нужно отсортировать список по невозрастанию (убыванию), то дополнительно прописывается параметр:

sorted(t, reverse=True)

Функции min(), max() и sorted() работают не только с числовыми типами, но и вообще с любыми, где допустимы операторы сравнения больше и меньше. Например, создадим список из символов (строк):

s = list("python")

И выполним все те же самые функции:

max(s)

min(s)

sorted(s)

А вот функция sum() приведет к ошибке:

sum(s)

## Практика. Перебор списков

1. Дан список. Вывести элементы списка.

Каждый элемент должен быть выведен с новой строки.

my\_list = [24, "Василий", "выхухоль", 42, -12.5]

2. Дан список. Вывести элементы списка, пронумеровав их начиная с единицы.

# Каждый элемент должен быть выведен с новой строки.

fruits = ["яблоко", "банан", "киви", "ананас", "груша"]

3. Сформировать список из n элементов, заполненный произвольными целыми числами в диапазоне от -10 до 10.

Вывести на экран сумму всех элементов данного списка.

## Операторы списков

При работе со списками часто используются следующие операторы:

* + – соединение двух списков в один;
* \* – дублирование списка;
* in – проверка вхождения элемента в список;
* del – удаление элемента списка.

Например:

[1, 2, 3] + [4, 5]

Но, вот так:

[1, 2, 3] + 4

работать не будет, так как оператор + соединяет именно списки между собой, просто число или какой-либо другой тип данных записывать нельзя. В данном случае правильно будет так:

[1, 2, 3] + [4]

Или, так:

[1, 2, 3] + [True]

Как видите, добавлять в список можно самые разные типы данных.

Следующий оператор \* выполняет дублирование списка указанное число раз:

["Я", "люблю", "Python"] \* 3

Этот оператор работает также как и со строками, здесь можно указывать только целое число (или переменную, ссылающуюся на целое значение). Прописывать дробные числа нельзя:

["Я", "люблю", "Python"] \* 3.5

На практике можно комбинировать операторы и определять, например, такие конструкции:

["Я"] + ["люблю"] \* 3 + ["Python"]

Все достаточно очевидно, гибко, наглядно и просто. Этим и знаменателен этот язык. В нем многое реализуется простыми и понятными методами, в отличие от других языков программирования.

Следующий оператор in позволяет определять вхождение некоторого значения в список. Делается это также просто, например:

lst = ["Москва", 1320, 5.8, True, "Тверь", False]

1320 in lst

120 in lst

Или, можно узнать, является ли элементом списка другой список:

[1, 2] in lst

В данном случае получим False, но если его добавить:

lst = ["Москва", 1320, 5.8, True, "Тверь", False, [1, 2]]

то

[1, 2] in lst

вернет True. И так со всеми типами данных:

"Москва" in lst

Последний оператор del выполняет удаление элемента списка по его индексу, например, так:

del lst[2]

при этом индексы оставшихся элементов по прежнему идут по порядку от 0 и до конца списка.

На этом мы завершим наше первое занятие по спискам. Из него вы должны себе хорошо представлять, как задавать списки с использованием квадратных скобок и функции list(), понимать механизм индексации (обращение к отдельным элементам списка), уметь оперировать функциями len(), max(), min(), sum(), sorted() и использовать операторы +, \*, in, del. Все это следует закрепить практическими заданиями, после чего, жду всех вас на следующем уроке.

## Срезы списков и сравнение списков

Мы продолжаем изучение списков языка Python. Это занятие начнем со срезов. О срезах мы с вами уже говорили, когда рассматривали строки и с их помощью выделяли наборы символов из строк. Со списками все работает аналогично: срезы позволяют выделять наборы элементов. Например, у нас есть список:

lst = ["Москва", "Уфа", "Тверь", "Казань"]

и мы хотим выбрать из него 2-й и 3-й элементы. Это можно сделать, так:

lst[1:3]

Давайте разберемся, как это работает. Синтаксис для срезов имеет вид:

список[start:end]

В данном случае, мы указываем стартовый индекс 1 (второй элемент) и конечный 3 (последний элемент, до которого выделяем срез, не включая его). В итоге, на выходе получаем список из двух элементов:

['Уфа', 'Тверь']

То есть, это новый список, состоящий из выделенных элементов. Мы в этом можем легко убедиться. Сохраним через переменную результат среза:

a = lst[1:3]

Затем, изменим его первый элемент:

a[0] = "Воронеж"

и, как видим, это никак не повлияло на исходный список lst.

В срезах можно указывать только начальный индекс:

lst[2:]

тогда все будет выбрано с 3-го элемента и до конца списка. Или только конечный индекс:

lst[:3]

Обратите внимание, последний указанный индекс не включается в срез. Если же не указывать ни первый, ни последний индексы, то получим копию исходного списка:

cities = lst[:]

Здесь создается именно копия списка. Если мы посмотрим на их id:

id(lst)

id(cities)

то они будут разными. Также копию списка, можно сделать с помощью рассмотренной ранее функции list:

c = list(lst)

Какой именно вариант использовать для копирования списка, решает сам программист, в зависимости от удобства. Чаще всего используется синтаксис срезов, т.к. это более короткая запись.

А что будет, если мы один список присвоим другому:

d = lst

будет ли здесь создаваться копия списка? Вспоминая занятие по переменным, мы говорили, что переменные – это ссылки на объекты и оператор присваивания лишь копирует ссылку, но не сам объект. Поэтому переменные d и lst будут ссылаться на один и тот же список, их id:

id(d)

id(lst)

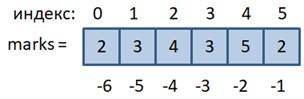
равные. Ну а наиболее сомневающимся предлагаю изменить значение списка через d:

d[0] = "Самара"

и посмотреть на список lst. В нем произойдет такое же изменение.

Далее, так как у списка имеются отрицательные индексы, то они также могут быть использованы при определении срезов. Например, создадим список оценок:

marks = [2, 3, 4, 3, 5, 2]



и запишем следующие срезы:

marks[2:-1]

marks[-3:-1]

Дополнительно в срезах можно указывать шаг перебора элементов, согласно синтаксису:

список[start:stop:step]

Например:

marks[1:5:2]

или без указания границ:

marks[:5:2]

marks[1::2]

marks[::2]

Если шагом является отрицательное число, то перебор элементов осуществляется с конца списка, например:

marks[::-1]

marks[::-2]

Вообще, механизм срезов работает абсолютно также как и со строками, только здесь они применяются к спискам.

Но, учитывая, что списки относятся к изменяемым типам данных, то со срезами можно выполнять одну дополнительную операцию – изменение группы элементов. Например, для списка marks мы хотим 3-ю и 4-ю оценки представить в виде строк. Это можно сделать, следующим образом:

marks[2:4] = ["хорошо", "удовлетв."]

И теперь коллекция содержит данные:

[2, 3, 'хорошо', 'удовлетв.', 5, 2]

Видите, как легко и просто это можно сделать. Или, другой пример, перебрать все элементы через один и присвоить им 0:

marks[::2] = [0, 0, 0]

Правда, такая конструкция содержит потенциальную ошибку. Если увеличить размер списка:

marks += [3]

и повторить операцию:

marks[::2] = [0, 0, 0]

то получим ошибку, так как число замен здесь уже четыре, а не три. Поэтому, в таких присваиваниях лучше явно указывать границы срезов:

marks[:5:2] = [0, 0, 0]

Также, для группового присваивания можно использовать и такой синтаксис:

marks[2:4] = 10, 20

## Сравнение списков

В заключение этого занятия рассмотрим возможности сравнения списков между собой с помощью операторов:

>, <, ==, !=

[1, 2, 3] == [1, 2, 3]   *# True*

[1, 2, 3] != [1, 2, 3]   *# False*

[1, 2, 3] > [1, 2, 3]    *# False*

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

[10, 2, 3] > [1, 2, 3]  *# True*

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]  *# False*

Все эти сравнения работают с однотипными данными:

[1, 2, "abc"] > [1, 2, "abc"]  *# False*

сработает корректно, а вот так:

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

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

На этом завершим очередное занятие. Надеюсь, вы теперь хорошо себе представляете механизм срезов, а также знаете, как выполняется сравнение списков между собой. После закрепления материала практическими заданиями, жду вас всех на следующем уроке.

Практика

1. Вводится строка с номером телефона в формате:

+7(xxx)xxx-xx-xx

Необходимо преобразовать ее в список lst (посимвольно, то есть, элементами списка будут являться отдельные символы строки). Затем, удалить первый '+', число 7 заменить на 8 и убрать дефисы. Отобразить полученный список на экране командой:

print("".join(lst))

**Sample Input:**

+7(912)123-45-67

**Sample Output:**

8(912)1234567

cities = input()

lst = list(cities)

del lst[0]

lst[0]='8'

lst.remove('-')

lst.remove('-')

print("".join(lst))

2. Строка содержит только заглавные буквы латинского алфавита (ABC…Z). Определите символ, который чаще всего встречается в файле между двумя одинаковыми символами.

Например, в тексте CBCABABACCC есть комбинации CBC, ABA (два раза), BAB и CCC. Чаще всего  — 3 раза  — между двумя одинаковыми символами стоит B, в ответе для этого случая надо написать B - 3.

Входные данные:

FPFHQDFWBPLLIJZPNKJEFWVHWNOKPRHUYVDALCXTQXHLSKTJGENYIDHSCPRBNUBMPICEHRRIQDSWDCWJFNQEPRFLDJFXSBUBECRZXHSPNPCQSACBBGMZBONQCKFFDAZBKJBTVCHMORBSFTQBHGDNDWXTEUQAHZDZXTOXHGIUKXMWBOQKTCYNCVPUWISKCNLENHFLQAIQAGNQUOVIFOBMNJOHFTRITCBZTWGVXSXOJUBZMABSGBUVXBKKPBZPVQHLBLTJVDMIVKWUNDWGNZXOOOWUFTSNNENKFTFQDTJACANYWCZQDUMBDNYZGCZUIQRIVXINGWJBQTXYQHWALCXISXPDJOXYLSQBOFWUETLQJUQHKFGNZMFIRRDHQTXYENGUVZEKWMDXAWPRNCPEUORSTADPRDTPQYFTBUHXWJYNESUMMNQFXIYXBCZEPZJXHGQBINTXHDL

Выходные данные:

N - 2

a=[]  
for i in range(len(f)-1):  
 if f[i-1]==f[i+1]:  
 a.append(f[i])  
  
print(Counter(a).most\_common()[0][0])

## Основные методы списков

На этом занятии мы свами познакомимся с основными методами, которые есть у списков. Что такое методы вам должно быть уже известно, об этом была речь, когда мы рассматривали методы строк. Кратко напомню, что список – это объект и с этим объектом связаны функции, которые и называются его методами.

list(s)

Преобразует объект s в список.

l.append(x)

Добавляет новый элемент x в конец списка l.

l.extend(t)

Добавляет новый список t в конец списка l.

l.count(x)

Определяет количество вхождений x в список l.

l.index(x [,start [,stop]])

Возвращает наименьшее значение индекса i, где s[i] == x.

Необязательные значения start и stop определяют индексы начального и конечого

элементов диапазона, где выполняется поиск.

l.insert(i,x)

Вставляет x в элемент с индексом i.

l.pop([i])

Возвращает i-й элемент и удаляет его из списка. Если индекс i не указан,

возвращается последний элемент.

l.remove(x)

Отыскивает в списке s элемент со значением x и удаляет его.

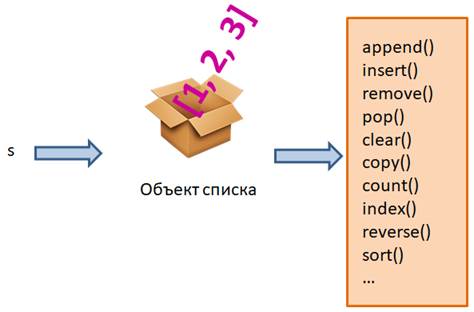
l.reverse()

Изменяет порядок следования элементов в списке l на обратный.

l.sort([key [, reverse]])

Сортирует элементы списка l. key – это функция, которая вычисляет значение

ключа. reverse – признак сортировки в обратном порядке.

Аргументы key и reverse всегда должны передаваться как именованные аргументы аргументы.

Давайте предположим, что у нас имеется список из чисел:

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

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

a.append(100)

Данный метод append ничего не возвращает, а меняет сам список. Поэтому писать здесь конструкцию вида:

a = a.append(100)

не следует, так как это приведет к потере данных. Этим методы списков отличаются от методов строк, когда мы записывали:

string="Hello"

string = string.upper()

Здесь метод upper возвращает измененную строку, поэтому все работает как и ожидается. А метод append ничего не возвращает, и присваивать значение None переменной a не имеет смысла.

Учитывая, что список может содержать самые разные данные, то в методе append можно прописывать не только число, но, 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. Например, вот так работать не будет:

a.append(1,2)

Если нам нужно вставить новый 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)

В этом можно убедиться по разным id этих объектов:

id(c)

id(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.

Ну и последний метод

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()

Здесь используется лексикографическое сравнение, о котором мы говорили, когда рассматривали строки.

Отличие метода sort() от ранее рассмотренной функции sorted() в том, что метод меняет сам список, а функция sorted() возвращает новый отсортированный список, не меняя начальный.

Если вам нужно расширить список другим списком, вы можете использовать метод extend, который добавляет переданный список в конец вашего списка.

collections = ['list', 'tuple', 'dict', 'set']

collections.extend(['ponyset', 'unicorndict'])

print(collections)

Также можно использовать перегруженный оператор +, который также добавляет переменную в конец вашего спиcка:

collections += [None]  
collections += [[1,2,3], (4,5,6), {7,8,9}, 10]  
print(collections)

Это все основные методы списков, которые вам следует знать и применять в практике программирования:

|  |  |
| --- | --- |
| 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 |

Как всегда, для закрепления материала обязательно пройдите практические задания.

## Практика методы списков

1. Даны два целых числа. Вывести список всех чисел кратных трем в диапазоне между заданными числами.

first\_number = int(input()) # Первое число

second\_number = int(input()) # Второе число

2. Дан список из элементов x = [-2,-1,0,1,2,3,4,5,6,7] – это координаты х, точек лежащих на прямой описываемой уравнением y=2x-4).

Получить и вывести на экран список координат y для данного уравнения прямой.

3. Пользователь вводит на экран дату в формате dd.mm.yyyy, например: 02.01.2013.

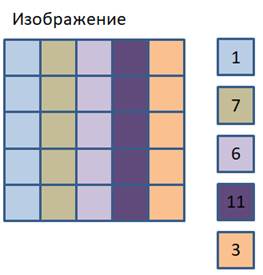
Ваша задача вывести дату в текстовом виде, например: второе января 2013 года.

Подсказка: создайте списки с названием дней и названиями месяцев. Примечание: для экономии времени, можно ограничить номер дня десятью. Примечание: склонениями названий можно пренебречь

data = '02.01.2013'  
data = list(map(int, data.split('.')))  
print(data)  
days = ['первое', 'второе', 'третье']  
mounts = ['января','февраля','марта']  
  
print(days[data[0]], mounts[data[1]], data[2], 'года')

# Вложенные списки, многомерные списки

Разберем, как формировать вложенные списки и работать с ними. Но сначала, что это такое и зачем они нужны.

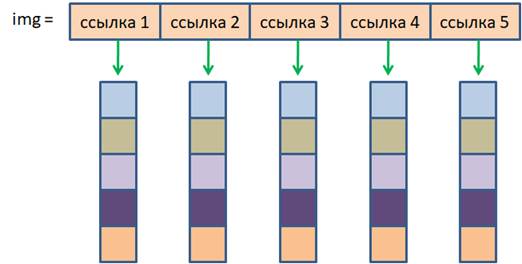


Давайте представим, что нам в программе нужно хранить изображение. Для примера я нарисовал его небольшим, всего 5 на 5 пикселей. Каждый цвет представляется своим уникальным числом. Я, условно, обозначил их 1, 7, 6, 11 и 3. Значит, для представления этих данных нам нужен двумерный список 5x5 с соответствующими числовыми значениями. Мы уже знаем, как задавать одномерный список:

line = [1, 7, 6, 11, 3]

Но так он описывает всего лишь одну строку. А нам нужно хранить пять таких строк. Учитывая, что элементом списка может быть другой список, то данное изображение можно задать так:

img = [[1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3]]



Мы здесь внутри первого списка определили пять вложенных и в результате получили двумерный список. Кстати, его можно было бы сформировать и проще, учитывая, что все вложенные списки одинаковы, на основе списка line, следующим образом:

img = [line[:], line[:], line[:], line[:], line[:]]

В итоге получим такой же список с независимыми строками:

[[1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3], [1, 7, 6, 11, 3]]

Вот эта последняя запись нам показывает структуру представления многомерных данных на уровне списков. Первый главный список хранит ссылки на вложенные списки. А вложенные списки уже хранят ссылки на соответствующие числа, представляющие тот или иной цвет. Поэтому, если взять первый элемент главного списка:

img0]

то мы получим список, представляющий первую строку (или, первый столбец в зависимости от интерпретации программистом этих данных). Главное, что мы получаем доступ к первому вложенному списку. А раз это так, то можно записать еще одни квадратные скобки и из этого вложенного списка взять, допустим, второй элемент:

img[0][1]

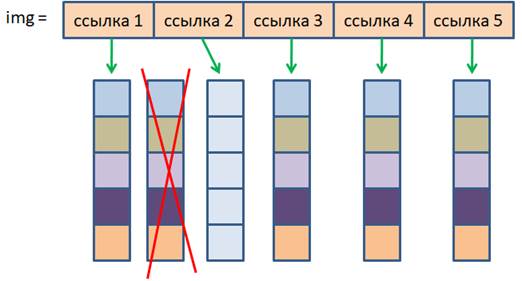
Также, можно заменить, например, вторую строку на новую, допустим, такую:

img[1] = [0, 0, 0, 0, 0]

или, то же самое, в более краткой форме:

img[1] = [0] \* 5

Что в итоге здесь произошло? Мы сформировали новый объект – список из нулей, связали с ним вторую ссылку главного списка, а прежний список был автоматически удален сборщиком мусора.



Если бы мы хотели изменить значения уже существующего вложенного списка, то следовало бы обратиться к его элементам, например, через механизм срезов:

img[1][:] = [0] \* 5

и присвоить его элементам новые числовые значения. Вот так это работает в деталях.

В качестве второго примера мы представим вложенными списками строки известного стихотворения на уровне отдельных слов:

Люблю тебя, Петра творенье,  
Люблю твой строгий, стройный вид,  
Невы державное теченье,  
Береговой ее гранит,

Здесь в каждой строке разное число слов, но для вложенных списков – это не проблема. Они могут иметь разное число элементов:

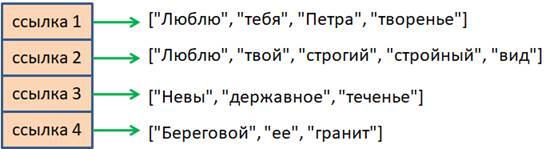
t = [["Люблю", "тебя", "Петра", "творенье"],

     ["Люблю", "твой", "строгий", "стройный", "вид"],

     ["Невы", "державное", "теченье"],

     ["Береговой", "ее", "гранит"]

     ]



В результате получаем следующую структуру наших данных. Здесь также для доступа к первой строке достаточно указать первый индекс:

t[0]

а к отдельному слову этой строки, второй индекс:

t[0][2]

Если же мы хотим изменить какое-либо слово, то это делается, следующим образом:

t[0][2] = "Питон"

В итоге, первая строка принимает вид:

['Люблю', 'тебя', 'Питон', 'творенье']

Мало того, мы можем добавить новую строку, используя известный метод:

t.append(["Твоих", "оград", "узор", "чугунный"])

Удалять список:

del t[1]

И так далее, то есть, делать с вложенными списками все те же операции, что и с обычными данными.

В заключение покажу пример многомерного списка с разными уровнями глубины:

A = [[[True, False], [1, 2, 3]], ["матрица", "вектор"]]

Смотрите, здесь

A[0]

это двумерный список, а

A[1]

одномерный вложенный список. Соответственно, для третьего уровня вложенности можем использовать три индекса для доступа к отдельному элементу, например:

A[0][1][0]

Здесь мы берем первый элемент, затем второй вложенный список и из него выбираем первый элемент.

Вот так в Python можно определять многомерные списки разной структуры и разного уровня вложенности.

# Генераторы списков (List comprehensions)

Мы продолжаем курс по языку программирования Python. На этом уроке мы с вами познакомимся с еще одной довольно полезной и популярной конструкцией – генератором списков. По-английски это записывается как:

List comprehension

Для лучшего понимания, давайте я начну сразу с конкретного примера. Предположим, мы хотели бы сформировать список из квадратов целых чисел от 0 до N. Используя наши текущие знания, очевидно, это можно было бы сделать, следующим образом:

N = 6

a = [0] \* N

for i in range(N):

    a[i] = i \*\* 2

print(a)

Но это не лучший вариант. Все то же самое можно реализовать буквально в одну строчку:

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

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.

Такой подход и называется генератором списков. Вначале указываем, как формируются значения списка, а затем, описываем изменение параметра x через ключевое слово for. Причем, здесь можно указывать только его. К примеру, ключевое слово while прописывать нельзя.

Итак, для генерации списков следует придерживаться следующего синтаксиса (определения):

[<способ формирования значения> for <переменная> in <итерируемый объект>]

В самом простом варианте мы можем сформировать список и так:

a = [1 for x in range(N)]

Хотя, в таких случаях, когда все значения одинаковы, обычно, используют оператор \*:

b = [1] \* N

Но, в отличие от простого дублирования с помощью List comprehension можно формировать более сложные последовательности, например, состоящих из остатков от деления на 4:

a = [x % 4 for x in range(N)]

То есть, мы можем использовать любые доступные нам операторы для формирования текущего значения. Если, например, взять оператор определения четного и нечетного значений:

a = [x % 2 == 0 for x in range(N)]

то получим список из булевых величин True и False. Или же сформировать значения линейной функции:

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

И так далее. Кроме того, мы здесь можем вызывать и функции. Пусть в программе пользователь вводит несколько целых чисел через пробел:

d\_inp = input("Целые числа через пробел: ")

Тогда с помощью генератора списков мы легко можем преобразовать их в набор чисел:

a = [d\_inp = input("Целые числа через пробел: ")

]

Здесь конструкция d\_inp.split() возвращает список строк из чисел, а функция int(d) преобразовывает каждую строку в целое число. Если убрать функцию int() и записать просто d:

a = [d for d in d\_inp.split()]

то у нас получится уже список из строк, то есть, функция int() действительно делает нужное нам преобразование.

Работа этой реализации очень похожа на функцию map(), которую мы с вами использовали в подобных задачах:

d\_inp = list(map(int, input("Целые числа через пробел: ").split()))

(О функции map() подробнее мы еще будем говорить).

Вообще, в генераторе списков можно использовать любые итерируемые объекты, в том числе и строки. Например, вполне допустима такая реализация:

a = [d for d in "python"]

получим список из отдельных символов. Или, можно каждый символ превратить в его код:

a = [ord(d) for d in "python"]

Имеем список из соответствующих кодов. Если же взять список из слов:

t = ["Я", "б", "Python", "выучил", "только", "за", "то", "что", "есть", "он", "на", "этом", "канале"]

то генератор списка:

a = [d for d in t]

сформирует точно такой же список. То есть, строки здесь воспринимаются уже как отдельные элементы и d ссылается на каждую из них по порядку. Я добавлю функцию len() для определения длины строк:

a = [len(d) for d in t]

и теперь мы имеем список из длин.

## Условия в генераторах списков

В генераторах дополнительно можно прописывать условия. Например, выберем все числа меньше нуля:

a = [x for x in range(-5, 5) if x < 0]

Мы здесь после оператора for записали необязательный оператор if. В результате, в список будут попадать только отрицательные значения x. Или, можно сделать проверку на нечетность:

a = [x for x in range(-5, 5) if x % 2 != 0]

Получаем только нечетные числа из диапазона [-5; 5). Также можно прописывать составные условия, например:

a = [x for x in range(-6, 7) if x % 2 == 0 and x % 3 == 0]

Выберем все числа, которые кратны 2 и 3 одновременно. Или, такой пример, выберем из списка городов только те, длина которых меньше семи:

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

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

Видите, как легко и просто можно реализовать такую задачу с помощью List comprehension.

Ну и я хочу отметить на этом занятии – это возможность использования тернарного условного оператора внутри генераторов списков. Так как первым элементом мы можем прописывать любые конструкции языка Python, то кто нам мешает сделать следующее. Пусть имеется список чисел:

d = [4, 3, -5, 0, 2, 11, 122, -8, 9]

и мы хотим преобразовать его в последовательность со словами «четное» и «нечетное». Сделать это можно как раз с помощью тернарного условного оператора:

a = ["четное" if x % 2 == 0 else "нечетное" for x in d]

Обратите еще раз внимание, что этот оператор не является какой-то особой частью синтаксиса генератора списка. Мы здесь всего лишь используем его как обычный оператор языка Python. Это такой же оператор как сложение, умножение и другие:

a = [x + 2 for x in d]

которые мы также можем использовать при формировании списков.

Полученная запись генератора списка с тернарным оператором получилась не очень читаемой. Это можно исправить, если записать все в отдельных строчках:

a = ["четное" if x % 2 == 0 else "нечетное"

     for x in d

     if x > 0

]

Кстати, здесь можно добавить еще и условие.

На этом мы завершим первое знакомство с генераторами списков. Для закрепления материала обязательно пройдите практические задания, после чего переходите к следующему уроку, где мы продолжим знакомиться с этой темой.

## Вложенные генераторы списков

Мы продолжаем тему генераторов списков языка Python. На прошлом занятии мы увидели, как можно формировать списки, используя конструкции:

[<способ формирования значения> for <счетчик> in <итерируемый объект>]

и

[<способ формирования значения>  
   for <счетчик> in <итерируемый объект>  
   if <условие>  
]

Но, в действительности, Python позволяет записывать любое число циклов for в генераторах списков. Здесь операторы if после циклов являются необязательными (мы их можем прописывать, а можем и пропускать).

Как всегда, работу таких вложенных циклов лучше всего увидеть на примере. В самом простом варианте, мы сформируем просто пары чисел в списке, следующим образом:

a = [(i, j) for i in range(3) for j in range(4)]

print(a)

Для большей наглядности запишем генератор списка в несколько строк:

a = [(i, j)

     for i in range(3)

     for j in range(4)

]

Вот отсюда уже гораздо лучше видно, что второй цикл вложен в первый. То есть, сначала при i = 0 отрабатывает внутренний цикл по j от 0 до 3. Затем, переходим к первому циклу, i увеличивается на 1 и внутренний цикл повторяется. В результате, получаем пары чисел:

[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3)]

То есть, здесь у нас происходит работа двух вложенных циклов. Дополнительно, мы можем указывать условия для каждого из них, например, так:

a = [(i, j)

     for i in range(3) if i % 3 == 0

     for j in range(4)

]

Или, у обоих вместе:

a = [(i, j)

     for i in range(3) if i % 3 == 0

     for j in range(4) if j % 2 != 0

]

Используя этот подход, мы можем, например, сформировать таблицу умножения:

a = [f"{i}\*{j} = {i\*j}"

     for i in range(1, 4)

     for j in range(1, 4)

]

Или, двумерный список превратить в одномерный:

matrix = [[0, 1, 2, 3],

          [10, 11, 12, 13],

          [20, 21, 22, 23]]

a = [x

     for row in matrix

     for x in row

]

Обратите внимание, мы во втором цикле используем переменную row из первого цикла. Это вполне допустимая операция, т.к. второй цикл вложен в первый и в нем доступны все переменные, объявленные ранее.

Давайте еще раз посмотрим на исходное определение генератора списка:

[<оператор> for <счетчик> in <итерируемый объект>]

Как я уже отмечал, в качестве оператора можно записывать любую конструкцию языка Python. А раз так, то кто нам мешает прописать здесь еще один генератор:

[[генератор списка]  
   for <переменная> in <итерируемый объект>  
]

В результате, получим вложенный генератор списка. Давайте посмотрим на примере, как это будет работать:

M, N = 3, 4

matrix = [[a for a in range(M)] for b in range(N)]

print(matrix)

Получаем двумерный список, вида:

[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2]]

Результат вполне очевиден. Смотрите, сначала отрабатывает первый (внешний) генератор списка и переменная b = 0. Затем, выполнение переходит к вложенному генератору, который выдает список [0, 1, 2]. Этот список помещается как первый элемент основного списка. Далее, снова отрабатывает первый генератор и b принимает значение 1. После этого переходим к вложенному генератору, который возвращает такой же список [0, 1, 2]. И так пока не закончится работа первого генератора. В итоге, видим список, в который вложены четыре других списка.

Где может пригодиться такой подход? Например, для изменения значений двумерного списка. Давайте предположим, что у нас есть вот такой список:

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

И мы хотим возвести все его значения в квадрат. Лучше всего это сделать именно через генератор, следующим образом:

A = [[x \*\* 2 for x in row] for row in A]

В первом генераторе происходит перебор строк (вложенных списков) матрицы A, а во вложенном генераторе – обход элементов строк матрицы. Каждое значение возводится в квадрат и на основе этого формируется текущая строка. Обратите внимание, что во вложении мы можем использовать переменные из внешнего генератора списка, в частности, переменную row, ссылающуюся на текущую строку матрицы A.

Другой пример – это транспонирование матрицы A (то есть, замена строк на столбцы) с использованием вложенных генераторов. Сделать это можно, следующим образом:

A = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]

A = [[row[i] for row in A] for i in range(len(A[0]))]

Поясню работу этой конструкции. Сначала значение i = 0, а переменная row[i] пробегает первые значения строк матрицы A. В результате формируется первая строка транспонированной матрицы. Далее, i увеличивается на 1 и row[i] пробегает уже вторые элементы строк матрицы A. Получаем вторую строку транспонированной матрицы. И так делаем для всех столбцов. На выходе формируется транспонированная матрица.

С таким типом вложений, я думаю, в целом все должно быть понятно. Другой вариант, когда мы список помещаем в качестве итерируемого объекта. Да, сам по себе генератор списка поддерживает механизм итерирования – перебора элементов через итератор, поэтому может быть использован в операторе for, например, так:

g = [u \*\* 2 for u in [x+1 for x in range(5)]]

Здесь сначала отрабатывает вложенный генератор списка, получаем список [1, 2, 3, 4, 5], а затем, эти значения перебираются первым генератором и возводятся в квадрат, получаем результат:

[1, 4, 9, 16, 25]

Это очень похоже на вычисление значений сложной (вложенной) функции:

g(u(x+1)) = (x+1) ^ 2

Вот такие варианты вложений генераторов списков, а также их комбинации, можно использовать в Python.

# Практика генераторы списков

**1.** Вводятся вещественные числа в строку через пробел. Необходимо на их основе сформировать список lst с помощью list comprehension (генератора списков) из модулей введенных чисел (в списке должны храниться именно числа, а не строки). Результат вывести на экран в виде списка командой:

print(lst)

**Входные данные:**

5.56 -8.7 1.0 3.14 77.845

**Выходные данные:**

[5.56, 8.7, 1.0, 3.14, 77.845]

**2.** Вводится семизначное целое положительное число. С помощью list comprehension сформировать список lst, содержащий цифры этого числа (в списке должны быть записаны числа, а не строки). Результат вывести на экран список командой:

print(lst)

**Входные данные:**

4567397

**Выходные данные:**

[4, 5, 6, 7, 3, 9, 7]

3. Вводятся названия городов в строку через пробел. Необходимо сформировать список с помощью list comprehension, содержащий названия длиной более пяти символов. Результат вывести в строчку через пробел.

Входные данные:

Казань Уфа Москва Челябинск Омск Тур Самара

Выходные данные:

Казань Москва Челябинск Самара

4. Вводится натуральное число n. Необходимо сформировать список с помощью list comprehension, состоящий из делителей числа n (включая и само число n). Результат вывести на экран в одну строку через пробел.

Входные данные:

10

Выходные данные:

1 2 5 10

5. Вводится натуральное число N. Необходимо сгенерировать вложенный список с помощью list comprehension, размером N x N, где первая строка содержала бы все нули, вторая - все единицы, третья - все двойки и так до N-й строки. Результат вывести в виде таблицы чисел как показано в примере ниже.

Входные данные:

4

Выходные данные:

0 0 0 0

1 1 1 1

2 2 2 2

3 3 3 3

5. Вводится список вещественных чисел. С помощью list comprehension сформировать список, состоящий из элементов введенного списка, имеющих четные индексы (то есть, выбрать все элементы с четными индексами). Результат вывести на экран в одну строку через пробел.

Входные данные:

8.5 11.3 1.0 -4.5 11.34 6.45

Выходные данные:

8.5 1.0 11.34

6. Вводятся два списка целых чисел одинаковой длины каждый с новой строки. С помощью list comprehension сформировать третий список, состоящий из суммы соответствующих пар чисел введенных списков. Результат вывести на экран в одну строку через пробел.

Входные данные:

1 2 3 4 5

6 7 8 9 10

Выходные данные:

7 9 11 13 15

7. Вводится список городов в одной строке. Необходимо с помощью list comprehension сформировать список lst, содержащий вложенные списки из пар:

<город> <численность населения>

Численность населения - целое число в тыс. человек. Вывести результат на экран

Входные данные:

Москва 15000 Уфа 1200 Самара 1090 Казань 1300

Выходные данные:

[['Москва', 15000], ['Уфа', 1200], ['Самара', 1090], ['Казань', 1300]]

# Комбинаторика. Модуль itertools

## Комбинаторика: сочетания

Модуль itertools позволяет решать программные задачи, построенные на структурах комбинаторики.

Сочетания – выбранные из множества n объектов комбинации m объектов, отличающиеся хотя бы одним объектом. Порядок элементов не важен.

Например, мы хотим составить трёхцветный флаг из лент цветных тканей. Есть четыре цвета лент. Все варианты выбора тканей без учёта их расположения:

colors = ['белый', 'жёлтый', 'синий', 'красный']

for item in itertools.combinations(colors, 3):

print(item)

Порядок следования не имеет значения, поэтому все тройки цветов уникальны.

## Комбинаторика: размещения

Размещения – те же сочетания, для которых важен порядок следования элементов. В продолжение предыдущего примера определим все варианты как мы можем составить флаг с учётом порядка следования цветов:

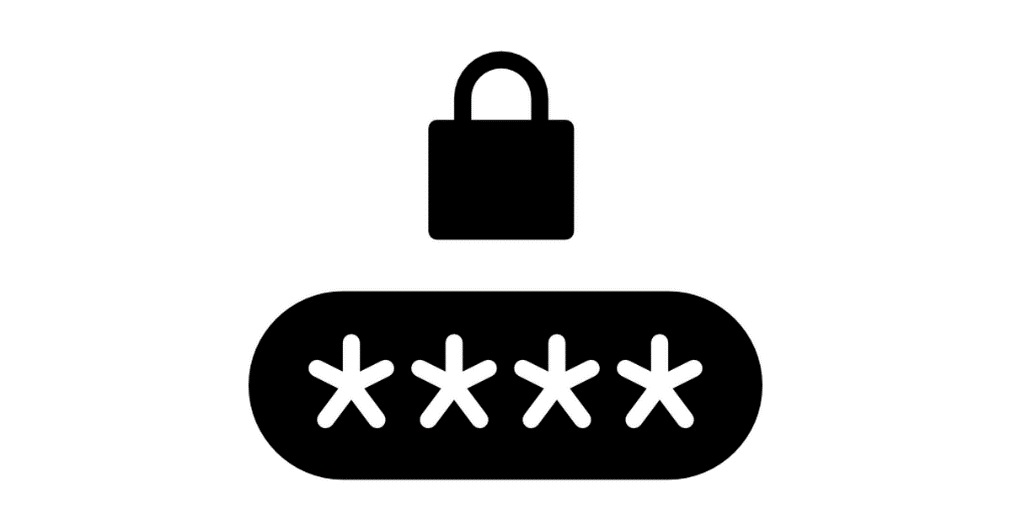
for item in itertools.permutations(colors, 3):

print(item)

Как и ожидалось, число комбинаций с учетом порядка элементов гораздо больше.

## Комбинаторика: размещение с повторениями

Размещение с повторениями (выборка с возвращением) – это комбинаторное размещение объектов, в котором каждый объект может участвовать в размещении несколько раз.



Например, есть пин-код из четырех цифр. На каждой позиции стоит цифра от 0 до 9. Позиции не зависят друг от друга. Переберем все возможные коды:

digits = range(10)

pincode\_vars = itertools.product(digits, repeat=4)

for var in pincode\_vars:

print(var)

(0, 0, 0, 0)

(0, 0, 0, 1)

(0, 0, 0, 2)

...

(9, 9, 9, 8)

(9, 9, 9, 9)

## Практика комбинаторика

Шифр кодового замка представляет собой последовательность из пяти символов, каждый из которых является цифрой от 1 до 4. Сколько различных вариантов шифра можно задать, если известно, что цифра 1 встречается ровно два раза, а каждая из других допустимых цифр может встречаться в шифре любое количество раз или не встречаться совсем?

from itertools import product

alphabet = '1234'

ap=[]

for i in product(alphabet, repeat=5):

if i.count('1') == 2:

ap.append(i)

print(len(ap))

Ответ: 270

Составляют 5-буквенные слова из букв слова ПЯТНИЦА. Найти количество слов, которые не начинаются с Н и в которых есть только одна буква Я. Буквы могут в слове могут повторяться.

from itertools import product

count = 0

for p in product("ПЯТНИЦА", repeat=5):

if p.count("Я") == 1 and p[0]!="Н":

count+=1

print(count)

Ответ: 5616

Сколько слов длины 6, начинающихся с согласной буквы, можно составить из букв Г, О, Д? Каждая буква может входить в слово несколько раз. Слова не обязательно должны быть осмысленными словами русского языка.

Ответ: 486.

import itertools

alphabet = "ГОД"

con = "ГД"

ar = itertools.product(alphabet, repeat=6) *#Размещение с повторением*

arl = []

for i in ar:

arl.append(list(i))

count = 0

for e in arl:

if e[0] in con:

count += 1

print(count)