**Кортежи и операции с ними**

***Кортеж****– это упорядоченная неизменяемая коллекция произвольных данных.*

Для задания кортежа достаточно перечислить значения через запятую: или же, это эквивалентно такой записи:

a = 1,2

a=(1,2,3)

Чтобы присвоить только одно значение кортежу, используется такой синтаксис: b = 1,     или    b = (1,) то есть, последняя запятая указывает, что 1 следует воспринимать как первый элемент кортежа. Если записать просто 1, то оно будет восприниматься как обычное число.

Для определения длины кортежа (числа его элементов), используется уже знакомая вам функция

len(a)

Доступ к элементам кортежа осуществляется также как и к элементам списков:

- по индексу:

a[0]

- через срезы:

a[1:2]

О срезах мы подробно говорили, когда рассматривали списки, поэтому не вижу смысла дублировать здесь эту информацию. Обратите лишь внимание на то, что вот такая операция не создает копию кортежа – это будут ссылки на один и тот же кортеж: увидим одно и то же значение id.

b = a[:]

**print**(id(a), id(b))

Если кортежи так похожи на списки, то в чем их преимущество перед ними? Другими словами: в каких случаях следует использовать кортеж вместо списка?

Во-первых, кортеж – это неизменяемый тип, то есть, мы в отличие от списков, не можем выполнять такие операции:

a[1] = 100

это приведет к ошибке. Значит, кортеж можно использовать, когда необходимо «запретить» программисту менять значения элементов списка.

Во-вторых, кортежи можно использовать в качестве ключей у словарей, например, так:

В-третьих, кортеж занимает меньше памяти, чем такой же список, например:

lst=[1,2,3]

t = (1,2,3)

**print**(lst.\_\_sizeof\_\_())

**print**(t.\_\_sizeof\_\_())

Как видите, размер кортежа меньше, чем списка. Здесь использован метод \_\_sizeof\_\_, который возвращает размер данных в байтах.

В-четвертых, операции с кортежами выполняются быстрее, чем со списками, например, перебор элементов или доступ к отдельному элементу.

Отсюда получается такой один общий, глобальный вывод: если мы работаем с неизменяемым упорядоченным списком, то предпочтительнее использовать кортеж.

Чтобы создать пустой кортеж можно использовать такие конструкции:

a = ()

b = tuple()

**print**(type(a), type(b))

Спрашивается: зачем создавать пустой кортеж, если он относится к неизменяемым типам. В действительности, мы лишь не можем менять уже существующие элементы в кортеже, но можем добавлять новые, используя оператор +: или для добавления в начало кортежа:

a = ()

a = a +  (1,)

a = (2,3) + a

Также можно добавить вложенный кортеж:

a += (("a", "hello"),)

или делать дубли в кортеже:

b = (0,)\*10

b = ("hello", "world")\*5

Все эти операции вполне допустимы.

Далее, функция tuple позволяет создавать кортежи на основе любых упорядоченных списков:

a = tuple([1,2,3])

a = tuple("Привет мир")

И обратно, из кортежа можно сформировать список:

t = (1,2,3)

lst = list(t)

Также кортежи могут хранить самые разные данные:

a = (True, [1,2,3], "hello", 5, {"house": "дом"})

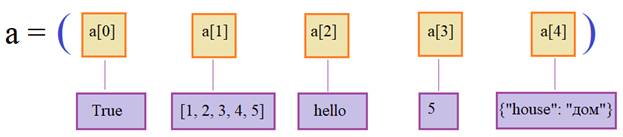
Причем, смотрите, если обратиться, например, к списку:

a[1]

то сам по себе список – это изменяемый тип, следовательно, его значение даже в кортеже мы можем спокойно менять:

a[1].append("5")

То есть, неизменяемость кортежа относится к его структуре элементов и переменным



которые ссылаются на конкретные объекты. Но, если объекты могут изменяться, то это никто не запрещает делать.

Так как сам кортеж – неизменяемый объект, то удалить отдельные его элементы невозможно: но можно удалить его целиком: после этого кортеж a перестанет существовать.

**del** a[1]

**del** a

**Методы кортежа**

У кортежа имеется два таких метода:

a = ("abc", 2, [1,2], True, 2, 5)

a.count("abc")

a.count(2)

который возвращает число найденных элементов с указанным значением. Если элемент не найден:

a.count("a")

то возвращается число 0.

Следующий метод:

a.index(2)

возвращает индекс первого найденного элемента с указанным значением.

Второй необязательный параметр

a.index(2,3)

позволяет задавать индекс начала поиска элемента. В данном случае поиск будет начинаться с третьего индекса.

А последний третий аргумент:

a.index(2, 3, 5)

определяет индекс, до которого идет поиск (не включая его). Если же записать вот так:

a.index(2, 3, 4)

то возникнет ошибка, т.к. в поиск будет включен только 3-й индекс и там нет значения 2.

**Особенности кортежей**

Кортежи поддерживают те же операции, что и списки, за исключением изменяющих содержимое.

Кортежи поддерживают:

* доступ к элементу по индексу (только для получения значений элементов);
* методы, в частности index(), count();
* встроенные функции, в частности len(), sum(), min() и max();
* срезы;
* оператор принадлежности in;
* операторы конкатенации (+) и повторения (\*).

## Вложенные кортежи

Подобно спискам, мы можем создавать вложенные кортежи.

Следующий программный код:

colors = ('red', ('green', 'blue'), 'yellow')

numbers = (1, 2, (4, (6, 7, 8, 9)), 10, 11)

print(colors[1][1])

print(numbers[2][1][3])

## Перебор кортежей

Перебор элементов кортежа осуществляется точно так же как перебор элементов списка. Для вывода **каждого** из элементов кортежа **на отдельной строке** можно использовать следующий код:

**Вариант 1.** Если нужны индексы элементов:

numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

for i in range(len(numbers)):

print(numbers[i])

**Вариант 2.** Если индексы не нужны:

numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

for num in numbers:

print(num)

Можно также использовать операцию **распаковки кортежа**.

numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

languages = ('Python', 'C++', 'Java')

print(\*numbers)

print(\*languages, sep='\n')

## Сравнение кортежей

Кортежи можно сравнивать между собой.

Приведенный ниже код:

print((1, 8) == (1, 8))

print((1, 8) != (1, 10))

print((1, 9) < (1, 2))

print((2, 5) < (6,))

print(('a', 'bc') > ('a', 'de'))

Обратите внимание: операции == и != применимы к любым кортежам, независимо от типов элементов. А вот операции <, >, <=, >= применимы только в том случае, когда соответствующие элементы кортежей имеют один тип.

Приведенный ниже код:

print((7, 5) < ('java', 'python'))

выводит:

TypeError: '<' not supported between instances of 'int' and 'str'

Сравнение кортежей происходит последовательно элемент за элементом, а если элементы равны — просматривается следующий элемент.

## Сортировка кортежей

Как мы помним, списки имеют метод sort(), который осуществляет сортировку на месте, то есть меняет порядок исходного списка. Поскольку кортежи по своей природе неизменяемы, то встроенного метода sort() они не содержат, тем не менее с помощью встроенной функции sorted() (не путать с списочным методом sort()) мы можем сортировать значения в кортежах.

not\_sorted\_tuple = (34, 1, 8, 67, 5, 9, 0, 23)

print(not\_sorted\_tuple)

sorted\_tuple = tuple(sorted(not\_sorted\_tuple))

print(sorted\_tuple)

Обратите внимание, что функция sorted() возвращает список, но с помощью функции tuple() мы приводим результат сортировки к кортежу.

## Преобразование кортежа в список и строку

Часто на практике нам приходится преобразовывать кортежи в списки и в строки. Для этого используются функции и методы str(), list(), tuple(), join().

### Преобразование кортежа в список и наоборот

**Кортеж можно преобразовать в список** с помощью функции list().

**Список можно преобразовать в кортеж** с помощью функции tuple().

**Кортеж можно преобразовать в строку** с помощью строкового метода join().

Приведенный ниже код:

notes = ('Do', 'Re', 'Mi', 'Fa', 'Sol', 'La', 'Si')

string1 = ''.join(notes)

string2 = '.'.join(notes)

print(string1)

print(string2)

Обратите внимание, что для применения строкового метода join() кортеж должен содержать именно строковые элементы. Если элементы кортежа отличны от строк, то требуется предварительно их преобразовать.

**Строку можно преобразовать в кортеж** с помощью функции tuple().

Приведенный ниже код:

letters = 'abcdefghijkl'

tpl = tuple(letters)

print(tpl)

выводит:

('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l')

Обратите внимание, что следующий код:

number = 12345

tpl = tuple(number)

print(tpl)

приведет к ошибке: TypeError: 'int' object is not iterable поскольку тип данных int не является итерируемым объектом. Для преобразования числа в кортеж, сначала нужно число преобразовать в строку, а уже потом использовать функцию tuple().

**Задания для самопроверки**

**Подвиг 1.** Вводятся названия городов в одну строку через пробел. На их основе формируется кортеж. Если в этом кортеже нет города "Москва", то следует его добавить в конец кортежа. Результат вывести на экран в виде строки с названиями городов через пробел.

**Подвиг 2.** Вводятся целые числа в одну строку через пробел. На их основе формируется кортеж. Необходимо создать еще один кортеж с уникальными (не повторяющимися) значениями из первого кортежа. Результат отобразите в виде списка чисел через пробел.

P. S. Подобные задачи решаются, как правило, с помощью множеств, но в качестве практики пока обойдемся без них.

**Подвиг 3.** Имеется двумерный кортеж, размером 5 x 5 элементов:

t = ((1, 0, 0, 0, 0),

     (0, 1, 0, 0, 0),

     (0, 0, 1, 0, 0),

     (0, 0, 0, 1, 0),

     (0, 0, 0, 0, 1))

Вводится натуральное число N (N < 5). Необходимо на основе кортежа t сформировать новый аналогичный кортеж t2 размером N x N элементов. Результат вывести на экран в виде таблицы чисел.

**Множества (set)**

***Множество (set)****– это неупорядоченная коллекция уникальных элементов.*

То есть коллекция, в которой отсутствуют дублирующие значения. Для создания множества используется следующий синтаксис: {значение1, значение2, …, значениеN}

Например: Выведем в консоль эту коллекцию и ее тип:

a={1,2,3,"hello"}

**print**(a, type(a))

Увидим: {1, 2, 3, 'hello'} <class 'set'>

Это объявление очень похоже на словарь. Только в словаре мы помимо значений еще указывали ключи. Здесь же ключей нет – только значения.

Причем, если мы попробуем добавить туда вот такие повторяющиеся значения: то они будут проигнорированы:

a={1,2,3,"hello", 2,3, "hello"}

И это ключевая особенность коллекции: она автоматически отбрасывает все дубли.

В качестве значений множества можно использовать любой неизменяемый тип: числа, строки, кортежи. Например:

a={1,1,"hi","hi",("a","b"), ("a","b")}

Также множества можно создавать с помощью специальной функции set:

b=set()

или, указать в качестве аргумента любой итерируемый объект:

b=set("hello world!")

b=set([1,1,2,3,5,3,2,1])

или даже функцию range:

b=set(range(7))

Обратите внимание, если попытаться создать пустое множество вот так:

b={}

то получим не пустое множество, а пустой словарь! Пустое множество создается именно с помощью функции set:

b=set()

Как можно использовать множества в программах?

Например, если нам нужно из списка удалить все дубли: мы можем его сначала превратить в множество из уникальных значений: а, затем, обратно в список, используя функцию list:

a=[1,2,3,43,2,4,3,2,1]

b=set(a)

a=list(b)

Видите, как эту операцию можно просто реализовать на Python. Причем, две последние операции можно объединить в одну вот так:

a=list(set(a))

Элементы множества можно обходить с помощью оператора цикла for:

setA = {7,6,5,4,3}

**for** x **in** setA:

**print**(x)

В результате, все значения множества будут выведены в консоль. Причем, мы можем перебирать именно значения множеств**, индексов у этих элементов нет**, т.к. множество – это неупорядоченная коллекция. И вот такая операция приведет к ошибке.

setA[0]

**Методы добавления/удаления элементов в множестве**

Для добавления элемента в множество используется метод **add**:

b.add(7)

И, так как множества – это изменяемый тип данных, то этот метод сразу добавит этот элемент, если такого еще нет. То есть, если мы попробуем добавить 7 еще раз: то множество не изменится.

Если необходимо в множество добавить сразу несколько значений, то для этого можно воспользоваться методом **update**:

b.update(["a", "b", (1,2)])

В качестве аргумента мы должны указать итерируемый объект, например, список, значения которого и будут добавлены в множество, разумеется, с проверкой их уникальности. Или, так:

b.update("abrakadabra")

Строка – это тоже итерируемый объект, и ее уникальные символы будут добавлены в множество. И так далее, в качестве аргумента метода update можно указывать любой перебираемый объект.

Для удаления элемента по значению используется метод **discard**:

b.discard(2)

Если еще раз попытаться удалить двойку: то ничего не произойдет и множество не изменится.

Другой метод для удаления элемента по значению – **remove**:

b.remove(4)

но при повторном таком вызове возникнет ошибка, т.к. значение 4 в множестве уже нет. То есть, данный метод возвращает ошибку при попытке удаления несуществующего значения. Это единственное отличие в работе этих двух методов.

Также удалять элементы можно и с помощью метода pop:

b.pop()

При этом он возвращает удаляемое значение, а сам удаляемый элемент оказывается, в общем-то, случайным, т.к. множество – это неупорядоченный список. Если вызвать этот метод для пустого множества, то возникнет ошибка:

c=set()

c.pop()

Наконец, если нужно просто удалить все элементы из множества, то используется метод:

b.clear()

# Операции над множествами

Для определения длины (числа элементов) множества используется функция len:

a={"abc", (1,2), 5, 4, True}

len(a)

Для проверки наличия значения в множестве используется оператор in:

"abc" **in** a

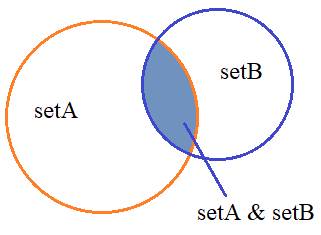
## Пересечение множеств

Для любых двух множеств: можно вычислять их пересечение, то есть, находить значения, входящие в состав обоих множеств. Это делается с помощью оператора &:

setA = {1,2,3,4}

setB = {3,4,5,6,7}

res = setA & setB



Здесь создается новое множество с соответствующими значениями. Сами исходные множества остаются без изменений. Если пересекающихся значений нет, то результатом будет пустое множество.

Этот оператор можно заменить эквивалентным методом intersection:

setA = {1,2,3,4}

setB = {3,4,5,6,7}

setA.intersection(setB)

который возвращает результат пересечения этих множеств. Сами же множества остаются без изменений. То есть, его обычно используют так:

res = setA.intersection(setB)

## Объединение множеств

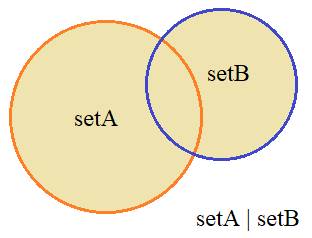
Противоположная операция – объединение двух множеств выполняется с помощью оператора | :

setA = {1,2,3,4}

setB = {3,4,5,6,7}

setA | setB

на выходе получим новое множество неповторяющихся значений из обоих множеств:



Эту же операцию можно записать и так:

setA |= setB

тогда на результат объединения будет ссылаться переменная setA.

Или же можно воспользоваться методом который возвращает множество из объединенных значений.

setA.union(setB)

## Вычитания множеств

Следующая операция – это вычитание множеств. Например, для множеств:

setA = {1,2,3,4}

setB = {3,4,5,6,7}

операция

setA - setB

возвратит новое множество, в котором из множества setA будут удалены все значения, существующие в множестве setB:

{1, 2}

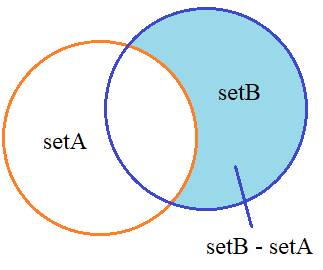
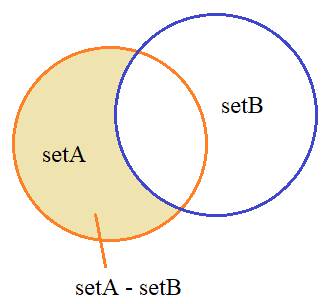
Или, наоборот, из множества setB вычесть множество setA:

setB – setA

получим значения

{5, 6, 7}

из которых исключены величины, входящие в множество setA.

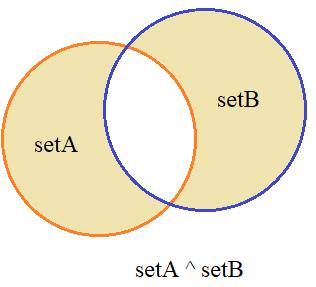


**симметричная разность**

setA = {1,2,3,4}; setB = {3,4,5,6,7}

setA ^ setB

то есть, множество, составленное из значений, не входящих одновременно в оба множества. В данном случае получим результат: {1, 2, 5, 6, 7}



## Сравнение множеств

Множества можно сравнивать между собой:

### **На равенство**

setA == setB

В данном случае получим False, т.к. множества не равны. Они считаются равными, если все элементы, входящие в одно множество, также принадлежат другому множеству и мощности этих множеств равны (то есть они содержат одинаковое число элементов). Например, такие:

setA = {7,6,5,4,3}; setB = {3,4,5,6,7}

тогда оператор

setA == setB

вернет значение True. Как видите, порядок элементов в множествах не играет роли при их сравнении.

### **На неравенство**

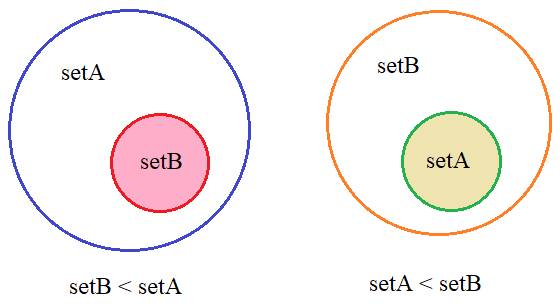
Противоположное сравнение на неравенство записывается так:

setA != setB

и возвращает True, если множества не равны и False, если равны.

### **На больше, меньше**

В Python операторы <, > применительно к множествам, по сути, определяют вхождение или не вхождение одного множества в другое. Математически, одно множество принадлежит (входит) другому, если все элементы первого множества принадлежат элементам второго множества:



Например, возьмем множества

setA = {7,6,5,4,3}; setB = {3,4,5}

тогда операция

setB < setA

вернет True, а операция

setA < setB

значение False. Но, если хотя бы один элемент множества setB не будет принадлежать множеству setA:

setB.add(22)

то обе операции вернут False.

Для равных множеств

setA = {7,6,5,4,3}; setB = {3,4,5,6,7}

обе операции также вернут False. Но вот такие операторы:

setA <= setB

setA >= setB

вернут True.

**Примечание** . Индексация и срезы недоступны для множеств.

**Примечание** . Операция конкатенации + и умножения на число \* недоступны для множеств.

**Примечание** . Если мы не изменяли множество, порядок обхода элементов при помощи цикла for не изменится.

**Примечание** . После изменения множества (методы add(), remove(), и т.д.) порядок элементов может измениться произвольным образом.

**Примечание** . Все основные операции над множествами выполнятся двумя способами: при помощи метода или соответствующего ему оператора. Различие в том, что метод может принимать в качестве аргумента не только множество (тип данных set), но и любой

mylist = [2021, 2020, 2019, 2018, 2017, 2016]

mytuple = (2021, 2020, 2016)

mystr = 'abcd'

myset = {2009, 2010, 2016}

print(myset.union(mystr)) # объединяем со строкой

print(myset.intersection(mylist)) # пересекаем со списком

print(myset.difference(mytuple)) # находим разность с кортежем

Приведенный ниже код:

print(myset | mystr)

print(myset & mylist)

print(myset - mytuple)

приводит к возникновению ошибок:

**Примечание** . Некоторые методы (union(), intersection(), difference()) и операторы (|, &, -, ^) позволяют совершать операции над несколькими множествами сразу.

Приведенный ниже код:

myset1 = {1, 2, 3, 4, 5, 6}

myset2 = {2, 3, 4, 5}

myset3 = {5, 6, 7, 8}

union1 = myset1.union(myset2, myset3)

union2 = myset1 | myset2 | myset3

difference1 = myset1.difference(myset2, myset3)

difference2 = myset1 - myset2 - myset3 # порядок выполнения слева-направо

print(union1 == union2)

print(difference1 == difference2)

**Примечание**. Оператор ^ симметрической разности позволяет использовать несколько множеств, а метод symmetric\_difference() – нет.

symdifference = myset1 ^ myset2 ^ myset3 # порядок выполнения слева-направо

print(symdifference)

symdifference = myset1.symmetric\_difference(myset2, myset3)

print(symdifference)

приводит к ошибке:

|  |  |
| --- | --- |
| A | B  A.union(B) | Возвращает множество, являющееся объединением множеств A и B |
| A |= B  A.update(B) | Добавляет в множество A все элементы из множества B |
| A & B  A.intersection(B) | Возвращает множество, являющееся пересечением множеств A и B |
| A &= B  A.intersection\_update(B) | Оставляет в множестве A только те элементы, которые есть в множестве B |
| A - B  A.difference(B) | Возвращает разность множеств A и B |
| A -= B  A.difference\_update(B) | Удаляет из множества A все элементы, входящие в B |
| A ^ B  A.symmetric\_difference(B) | Возвращает симметрическую разность множеств A и B |
| A ^= B  A.symmetric\_difference\_update(B) | Записывает в A симметрическую разность множеств A и B |

Приоритет операторов в порядке убывания (верхние операторы имеют более высокий приоритет, чем нижние) имеет вид:

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| - | разность |
| & | пересечение |
| ^ | симметрическая разность |
| | | объединение |

# Генераторы множеств, frozenset

Пусть требуется создать множество, содержащее цифры введенного числа.

Следующий программный код: digits = set(int(input())) приводит к ошибке поскольку функция set() принимает в качестве аргумента итерируемый объект, а число (тип данных int) таковым не является.

Следующий программный код: digits = set(input())

при вводе строки '12345' создает множество символов {'1', '2', '3', '4', '5'}, а не множество цифр {1, 2, 3, 4, 5}.

Для создания требуемого множества, содержащего уникальные цифры введенного числа, нам придется написать код:

digits = set()

for c in input():

digits.add(int(c))

Такой код хоть и не сложен, однако достаточно громоздок. Для создания множеств в Python можно использовать специальный синтаксис, как при создании [списка](https://stepik.org/lesson/326725/step/1?unit=310010).

Приведенный выше код можно переписать с использованием генератора множеств:

digits = {int(c) for c in input()}

Общий вид генератора множеств следующий: {выражение for переменная in последовательность}где  выражение — некоторое выражение, как правило, зависящее от использованной в списочном выражении переменной, которым будут заполнены элементы множества переменная — имя некоторой переменной, последовательность — последовательность значений, которые она принимает (любой итерируемый объект).

### **Примеры использования генератора множеств**

**1. Создать множество, заполненное квадратами целых чисел от 0 до 9 можно так:**

squares = {i \*\* 2 for i in range(10)}

**2. Создать множество, заполненное кубами целых чисел от 10 до 20 можно так:**

cubes = {i \*\* 3 for i in range(10, 21)}

**3. Создать множество, заполненное символами строки можно так:**

chars = {c for c in 'abcdefg'}

В генераторах множеств можно использовать условный оператор. Например, если требуется создать множество, заполненное только цифрами некоторой строки, то мы можем написать такой код:

digits = {int(d) for d in 'abcd12ef78ghj90' if d.isdigit()}

**Неизменяемые множества**

Неизменяемые множества frozenset представляют собой те же самые множества set, за исключением того, что set – это изменяемый тип данных, а frozenset – неизменяемый тип данных. Подобная ситуация была со списками и кортежами, где списки – изменяемый тип данных, а кортежи – неизменяемый.

Неизменяемые множества создаются при помощи функции frozenset() из любого итерируемого объекта. Причем множества, созданные из одного и того же итерируемого объекта при помощи set() и frozenset(), будут совпадать, но типы данных все же будут разные.

a = set('Python')

b = frozenset('Python')

print(a == b)

print(type(a))

print(type(b))

Над неизменяемыми множествами frozenset определены все функции множеств set, которые не изменяют непосредственно само множество. Если попытаться добавить в созданное неизменяемое множество какой-либо элемент, в результате будет выдана ошибка.

* **Тип frozenset является НЕ изменяемым и хешируемым множеством**, его содержимое не может быть изменено после его создания, поэтому он может использоваться как ключ словаря или как элемент другого множества.

**Подвиг**

Вводится строка, содержащая латинские символы, пробелы и цифры. Необходимо выделить из нее все неповторяющиеся цифры (символы от 0 до 9) и вывести на экран в одну строку через пробел их в порядке возрастания значений. Если цифры отсутствуют, то вывести слово НЕТ.

Sample Input: Python 3.11.2 - best language!

Sample Output: 1 2 3

**Подвиг**

Вводятся два списка городов каждый с новой строки (в строке названия через пробел), которые объехал Сергей в 1-й и 2-й годы своего путешествия по России.

Требуется определить, включал ли его маршрут во 2-й год все города 1-го года путешествия? Если это так, то вывести ДА, иначе - НЕТ.

**Sample Input:**

Москва Казань Самара Москва

Москва Владимир Новгород Казань Самара Москва

**Sample Output:** ДА

**Подвиг**

Вводятся два списка целых чисел каждый с новой строки (в строке наборы чисел через пробел). Необходимо выбрать и отобразить на экране уникальные числа, присутствующие в первом или втором списках, но отсутствующие одновременно в обоих.

Результат выведите на экран в виде строки чисел, записанных по возрастанию через пробел.

**Sample Input:**

1 2 3 4 5

4 5 6 7 8

**Sample Output:**

1 2 3 6 7 8