Модуль 5. Кортежи, словари и множества

Введение

Повтор предыдущий материал

```
import time
from random import *


N = [1 for i in range(5)]
print(N)

seed(1) # хотите я угадаю вашу случайную (псевдо) последовательность

N = [random() for i in range(5)]
print(N)
```

Она при каждом вызове выдает случайное значение в диапазоне [0.0; 1.0). Обратите внимание, что числа подчиняются равномерному закону распределения, то есть, в диапазоне [0.0; 1.0) они могут принимать любое значение с равной вероятностью.

```
# равномерному закону распределения
N = [randint(0,5) for i in range(5)]
print(N)

# uniform(a, b) также генерирует случайные значения по равномерному закону
N= [uniform(0,10) for i in range(5)]
print(N)

N= [gauss(1.0,0.5) for i in range(5)]
print(N)

# Однако часто на практике требуются случайные величины с другим законом распределения – гауссовским (или, еще говорят, нормальным)
ему часто подчиняются многие реальные события: колебания цен на нефть,  и различных товаров, шумы при радиопередачах, погрешности измерений различных значений и тп

N= [randrange(0,100,10) for i in range(5)]
print(N)

N = sample(N,3)
print(N)

while True:
    shuffle(N)
    print(N)
    time.sleep(1)
```

Задачи на случайные последовательности

Задача 1:

Напишите программу, которая с помощью модуля random генерирует случайный пароль. Программа принимает на вход длину пароля и выводит случайный пароль, содержащий только символы английского алфавита a..z, A..Z (в нижнем и верхнем регистре). Применить генератор списка.

Задача 2. Дополнить задачу. В каждом пароле обязательно должна присутствовать хотя бы одна цифра и как минимум по одной букве в верхнем и нижнем регистре.

Задача 3: IP адрес состоит из четырех чисел из диапазона от 0 до 255 (включительно), разделенных точкой.

Напишите программу, которая с помощью модуля random генерирует и возвращает случайный корректный IP адрес.

4. Напишите программу, которая с помощью модуля random генерирует 20 случайных номеров лотерейных билетов и выводит их каждый на отдельной строке. Обратите внимание, вы должны придерживаться следующих условий:

- номер не может начинаться с нулей;
- номер лотерейного билета должен состоять из 7 цифр;
- все 10 лотерейных билетов должны быть различными.

У вас есть список слов: “книга”, “месяц”, “ручка”, “шарик”, “олень”, “носок”.

Необходимо выбрать случайным образом слово и отрисовать его, используя вместо букв какие-либо символы, например '\u25A0' (■), ‘\u2660' (♠) или '\u2665' (♥) Необходимо установить счётчик “жизни” в какое-либо значение, например 5

- Предложить игроку ввести букву или всё слово целиком.
- Если буква правильная, то слово перерисовывается с видимой буквой.
- Если буква неправильная, то у игрока отнимается одна “жизнь”.
- Если игрок ввёл слово и это слово правильно, либо это последняя правильная буква, либо у игрока закончились “жизни”, то игра заканчивается.

Примерный вид программы

```
■ ■ ■ ■ ■ | ❤x3
Назовите букву или слово целиком: е
■ ■ Е ■ ■ | ❤x3
Назовите букву или слово целиком: р
Неправильно. Вы теряете жизнь
■ ■ Е ■ ■ | ❤x2
Назовите букву или слово целиком: о
О ■ Е ■ ■ | ❤x2
Назовите букву или слово целиком: в
Неправильно. Вы теряете жизнь
О ■ Е ■ ■ | ❤x1
Назовите букву или слово целиком: ь
О ■ Е ■ Ь | ❤x1
Назовите букву или слово целиком: олень
О Л Е Н Ь
Вы выиграли! Приз в студию!
```

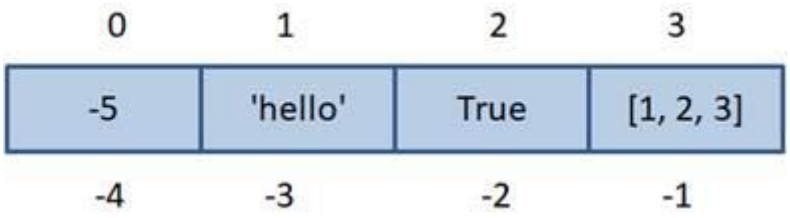
```
import random

s = ['книга', 'месяц', 'ручка', 'шарик', 'олень', 'носок']

random_word = random.choice(s)
base_word = ['\u25A0'] * 5
lifes = 3

print(*base_word, ' \u2665 ' * lifes)

while 1:
    attempt = input('Назовите букву или слово целиком -> ')
    if attempt in random_word:
        ind = random_word.index(attempt)
        base_word[ind] = attempt
    else:
        print('Неправильно. Вы теряете жизнь')
        lifes -= 1

    print(*base_word, ' \u2665 ' * lifes)
    if ''.join(base_word) == random_word:
        break

print('Ты угадал слово')
```

Кортежи (tuple) и их методы

На этом занятии мы с вами познакомимся с еще одной новой коллекцией данных – кортежами. Что такое кортеж? Это упорядоченная, но неизменяемая (immutable) коллекция произвольных данных.

Когда вы присваиваете кортежу элемент, он «запекается» и больше не изменяется.

Они, в целом, аналогичны спискам, но в отличие от них, элементы кортежей менять нельзя – это неизменяемый тип данных. Т.е. мы не можем добавлять или удалять элементы в кортеже, изменять его



Для задания кортежа достаточно перечислить значения через запятую:

```
a = 1,2
```

или, что то же самое, в круглых скобках:

```
a = (1, 2, 3)
```

Раньше мы уже работали с кортежем,

```
a, b, c = 3, 'Hello', True
print(a,b,c)
```

Под капотом, справа получался кортеж, который потом парсился в 3 переменные.

Но, обратите внимание, при определении кортежа с одним значение, следует использовать такой синтаксис:

b = 1, или b = (1,)

Здесь висячая запятая указывает, что единицу следует воспринимать как первый элемент кортежа, а не как число 1. Без запятой – это будет уже просто число один. Вот это нужно очень хорошо запомнить, начинающие программисты здесь часто делают ошибку, не прописывая висячую запятую.

Далее, мы можем переменным сразу присвоить соответствующие значения из кортежа, перечисляя их через запятую:

```
x, y = (1, 2)
```

но, если число переменных не будет совпадать с числом элементов кортежа:

```
x, y = (1, 2, 3)
```

то возникнет ошибка.

Длину кортежа (число его элементов) можно определить с помощью известной уже вам функции:

```
len(a)
```

А доступ к его элементам осуществляется также как и к элементам списков:

- по индексу:

  a[0] a[2] a[-1]

- через срезы:

  a[1:2] a[0:-1] a[:3] a[1:] a[:]

О срезах мы с вами уже подробно говорили, когда рассматривали списки, здесь все абсолютно также, кроме одной операции – взятия полного среза:

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

В случае с кортежем здесь не создается его копии – это будет ссылка на тот же самый объект:

```
print(id(a), id(b))
```

Напомню, что для списков эта операция приводит к их копированию. Этот момент нужно также очень хорошо запомнить: для кортежей – возвращается тот же самый объект, а для списков – создается копия.

Некоторые из вас сейчас могут задаваться вопросом: зачем было придумывать кортежи, если списки способны выполнять тот же самый функционал? И даже больше – у них можно менять значения элементов, в отличие от кортежей? Да, все верно, по функциональности списки шире кортежей, но у последних все же есть свои преимущества.

Во-первых, то, что кортеж относится к неизменяемому типу данных, то мы можем использовать его, когда необходимо «запретить» программисту менять значения элементов списка. Например, вот такая операция:

```
a[1] = 100
```

приведет к ошибке. Менять значения кортежей нельзя. Во-вторых, кортежи можно использовать в качестве ключей у словарей, например, так:

```
d = {}
d[a] = "кортеж"
```

Напомню, что списки как ключи применять недопустимо, так как список – это изменяемый тип, а ключами могут быть только неизменяемые типы и кортежи здесь имеют преимущество перед списками.

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.

Из всего этого можно сделать такой общий вывод: если мы работаем с неизменяемым упорядоченным списком, то предпочтительнее использовать кортеж.

Кортежи быстрее работают. По причине неизменяемости кортежи хранятся в памяти особым образом, поэтому операции с их элементами выполняются заведомо быстрее, чем с компонентами списка.

Теперь, когда мы поняли, что кортежи играют свою значимую роль в программах на Python, вернемся к их рассмотрению. Чтобы создать пустой кортеж можно просто записать круглые скобки:

```
a = ()
```

или воспользоваться специальной встроенной функцией:

```
b = tuple()
print(type(a), type(b))
```

Но здесь сразу может возникнуть вопрос: зачем создавать пустой кортеж, если он относится к неизменяемым типам данных? Слово неизменяемый наводит на мысль, что вид кортежа остается неизменным. Но это не совсем так. В действительности, мы лишь не можем менять уже существующие элементы в кортеже, но можем создавать новые, используя оператор +, например:

```
a = ()
a = a +  (1,)
```

или для добавления данных в начало кортежа:

```
a = (2, 3) + a
```

Также можно добавить вложенный кортеж:

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

или дублировать элементы кортежа через оператор *, подобно спискам:

```
b = (0,)*10
b = ("hello", "world") * 5
```

Как видите, все эти операции вполне допустимы. А вот удалять отдельные его элементы уже нельзя. Если мы попытаемся это сделать:

```
del a[1]
```

то возникнет ошибка. Правда, можно удалить объект целиком:

```
del a
```

тогда кортеж просто перестанет существовать, не будет даже пустого кортежа, здесь объект удаляется целиком.

Далее, с помощью функции tuple() можно создавать кортежи на основе любого итерируемого объекта, например, списков и строк:

```
a = tuple([1,2,3])
a = tuple("Привет мир")
print(f'{a = }')
```

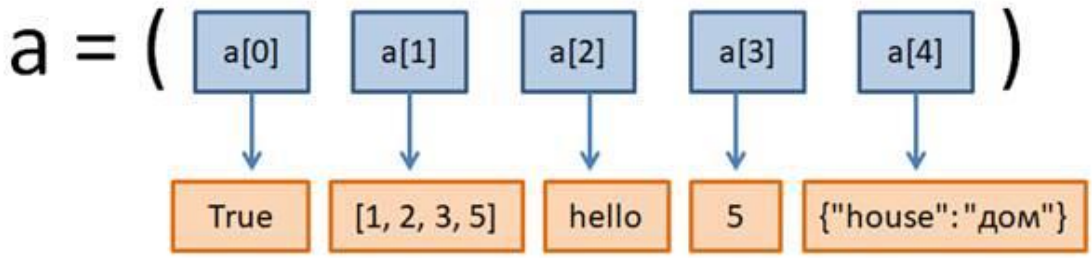
А если нам нужно превратить кортеж в список, то подойдет известная нам функция list(), которая формирует список также из любого итерируемого объекта:

```
t = (1,2,3)
lst = list(t)
```

Так как кортеж – это перебираемый, то есть, итерируемый объект, то никто не мешает нам превращать его в список.

Как я отмечал в самом начале, 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 это изменяемый объект, следовательно, его значение даже в кортеже мы можем спокойно менять:

```
a[1].append("5")
```

То есть, неизменность кортежа относится к его структуре элементов и значениям ссылок на объекты. Но, если сами объекты, при этом, могут меняться, то мы можем легко это делать. То есть, обращаясь ко второму элементу кортежа, мы, фактически, имеем список, с которым возможны все его операции без каких-либо ограничений. Я, думаю, это должно быть понятно?

В заключение рассмотрим два метода, которые часто используются в кортежах – это:

- tuple.count(значение) – возвращает число найденных элементов с указанным значением;
- tuple.index(значение[, start[, stop]]) – возвращает индекс первого найденного элемента с указанным значением (start и stop – необязательные параметры, индексы начала и конца поиска).

Эти методы работают так же, как и у списков. Пусть у нас есть кортеж:

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

И мы хотим определить число элементов со значениями:

```
a.count("abc")
a.count(2)
```

Как видите, метод count() возвращает число таких элементов в кортеже. Если же элемент не будет найден:

```
a.count("a")
```

то возвращается 0. Следующий метод:

```
a.index(2)
```

возвращает индекс первого найденного элемента с указанным значением. Если значение не найдено:

```
a.index(3)
```

то этот метод приводит к ошибке. Поэтому, прежде чем его использовать, лучше проверить, а есть ли такое значение вообще в кортеже:

```
3 in a
```

Второй необязательный параметр

```
a.index(2, 3)
```

позволяет задавать индекс начала поиска. 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.

Кроме того, следует вспомнить функцию enumerate

```
s = 'Python'

for item in enumerate(s):
    print(item)
```

которая также возвращает кортеж, который в свою очередь легко распаковывается в 2 переменные.

```
s = 'Python'

for item in enumerate(s):
    print(item)
```

Надеюсь, вы узнали, что такое кортежи, как с ними можно работать и зачем они нужны. В целом, вы должны запомнить следующее: операторы работы с кортежами, а также функции и методы кортежей. Закрепим этот материал практическими заданиями.

Вычислительная сложность структур данных

Вычислительная сложность это область из информатики, которая анализирует алгоритмы на основе количества ресурсов, необходимых для его запуска, Количество требуемых ресурсов зависит от размера входных данных и не только.

Рассмотрим пример

```
import platform
print (platform.processor() , platform.architecture())

import time

start = time.time()
x = [i for i in range(10_000_000)]
end = time.time()
print(end - start)

start = time.time()
x = (i for i in range(10_000_000))
```

```
end = time.time()
print(end - start)

start = time.time()
x = {i for i in range(10_000_000)}
end = time.time()
print(end - start)

start = time.time()
x = range(10_000_000)
end = time.time()
print(end - start)
```

Результат:

Intel64 Family 6 Model 142 Stepping 10, GenuineIntel ('64bit', 'WindowsPE')

0.695237398147583

0.08672451972961426

0.5249037742614746

0.12475299835205078

При работе с современными языками, такими как Python, который предоставляет встроенные функции, такие как алгоритмы сортировки, когда-нибудь вам, вероятно, потребуется реализовать алгоритм для выполнения некоторой операции с определенным объемом данных.

Изучая сложность времени, вы поймете важную концепцию эффективности и сможете найти узкие места в вашем коде, которые следует улучшить, главным образом при работе с огромными наборами данных.

Распаковка последовательностей

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

print (*a, sep=',')

s1,s2,s3,s4,s5 = 'hello'

print(s1)
s1,s2,s3,*s4 = 'hello'

print(s4) #['l', 'o']
```

```
s = 'hello python'
a, b = s.split()
print(b, a)

R = range(4,7)
a,b,c = R
print(a)
```

Звездочка может быть

```
a, *b, c = s

print(b)
```

```
s1, *s2, s3 = 'hello'

print(s2) #['e', 'l', 'l']
```

```
s = 'hello python'

_, _, c, *d, e = s

print(d)
```

```
s = '1234' # Какое минимальное количество элементов

_, _, c, *d, e = s

print(d)
```

Задания на кортежи

Задание 1. Имеется кортеж:

```
t = (3.4, -56.7)
```

Вводится последовательность целых чисел в одну строчку через пробел. Необходимо их добавить в кортеж t. Результат вывести на экран командой:

```
print(t)
```

Входные данные:

8 11 -5 2

Выходные данные:

(3.4, -56.7, 8, 11, -5, 2)

```
t = (3.4, -56.7)
s = tuple(map(int, input().split()))
t = t + tuple(s)
print (t)
```

Задание 2. 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.

Входные данные:

Уфа Казань Самара

Выходные данные:

Уфа Казань Самара Москва

```
values_users = tuple(input().split())
values_users += tuple('' if "Москва" in values_users else ("Москва",))
# Добавляем в наш кортеж Кортеж с слово "Москва"
# через тернарный оператор, если слово есть то не добавляем ни чего, если нет то добавляем слово "Москва"
print(*values_users)
```

Задание 3. Вводятся названия городов в одну строку через пробел. На их основе формируется кортеж. Если в этом кортеже присутствует город "Ульяновск", то этот элемент следует удалить (создав новый кортеж). Результат вывести на экран в виде строки с названиями городов через пробел.

Входные данные:

Воронеж Самара Тольятти Ульяновск Пермь

Выходные данные:

Воронеж Самара Тольятти Пермь

```
tup_city = tuple([i for i in values_users if i != 'Ульяновск'])
```

Задание 5. Вводятся имена студентов в одну строчку через пробел. На их основе формируется кортеж. Отобразите на экране все имена из этого кортежа, которые содержат фрагмент "ва" (без учета регистра). Имена выводятся в одну строчку через пробел в нижнем регистре (малыми буквами).

Входные данные:

Петя Варвара Венера Василиса Василий Федор

Выходные данные:

варвара василиса василий

```
values_users = tuple(input().lower().split()) # Создаем кортеж из пользовательских данных,
# разбивая на отдельные слова по пробелу с помощью метода .split()
# Так как КОРЕЖ неизменяемый тип данных мы можем только объединить кортежи
# if 'Ульяновск' in values_users:
tup_city = ()
tup_city += tuple([i for i in values_users if 'ва' in i])
```

Задание 6. 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.

Входные данные:

8 11 -5 -2 8 11 -5

Выходные данные:

8 11 -5 -2

```
values_users = tuple(input().lower().split()) # Создаем кортеж из пользовательских данных,
# разбивая на отдельные слова по пробелу с помощью метода .split()
# Так как КОРТЕЖ неизменяемый тип данных мы можем только объединить кортежи или создать новый
tup_city = () # Создаю временный кортеж
for i in range(0, len(values_users)): # Прохожу по кортежу
```

```
if values_users.count(values_users[i]) > 1 : # С помощью метода .count() считаю сколько одинаковых объектов
# в кортеже, если больше 1 то тогда сохраняю его индекс в кортеже
tup_city += (values_users.index(values_users[i],i),) # Сохраняю с помощью метода .index(values_users[i],i)
# Где первый аргумент это значение которое ищем, второе с какой позиции ищем.
print(*tup_city)
```

Задание 7. Вводятся целые числа в одну строку через пробел. На их основе формируется кортеж. Необходимо найти и вывести все индексы неуникальных (повторяющихся) значений в этом кортеже. Результат отобразите в виде строки чисел, записанных через пробел.

Входные данные:

5 4 -3 2 4 5 10 11

Выходные данные:

0 1 4 5

```
s = input()
t = tuple(map(int,s.split()))
for n,i in enumerate(t):
    if t.count(i)>1:
        print (n, end=' ')
```

Задание 8. 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.

Входные данные:

3

Выходные данные:

```
1 0 0
0 1 0
0 0 1
```

```
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))

N = int(input())
t2 = ()
for i in range(N):
    t2 += t[i][:N],

for i in t2:
     print (*i)
```

или

```
t = tuple(map(tuple, __import__('numpy').array(t)[0:n, 0:n]))
print(*[' '.join(map(str, row)) for row in t], sep='\n')
```

Задание 9. Вводятся пункты меню (каждый пункт с новой строки) в формате:

название_1 URL-адрес_1
название_2 URL-адрес_2
...
название_N URL-адрес_N

Необходимо эту информацию представить в виде вложенного кортежа menu в формате:

((название_1, URL-адрес_1), (название_2, URL-адрес_2), ... (название_N, URL-адрес_N))

Результат вывести на экран в виде кортежа командой:

```
print(menu)
```

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Входные данные:

Главная home

Python learn-python

Java learn-java

PHP learn-php

Выходные данные:

(('Главная', 'home'), ('Python', 'learn-python'), ('Java', 'learn-java'), ('PHP', 'learn-php'))

```
tup_end = tuple(tuple(i.split()) for i in lst_in)
```

Функции all() и any()

На этом занятии мы поговорим о функциях all() и any(). Первым делом посмотрим, что они делают и зачем нужны? Начнем с функции all() и очень простого примера. Предположим, что есть список булевых значений:

```
a = [True, True, True, True]
```

И мы хотим узнать, принимают ли все они значение True? Для этого, как раз и используется функция all(), которая на вход принимает итерируемый объект и все его значения приводит к булевым величинам:

```
all(a)
```

Если все значения равны True, то на выходе получаем True. Если же, хотя бы одно значение принимает False, то на выходе будет False:

```
all([True, True, False, True])
```

Но, как вы понимаете, значения в списках могут быть любыми, например, такими:

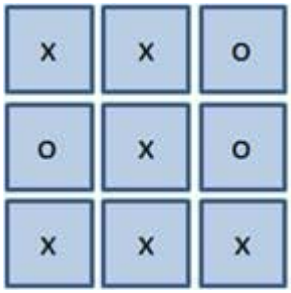
```
a = [0, 1, 2.5, "", "python", [], [1, 2], {}]
```

И к нему тоже можно применить функцию all():

```
all(a)
```

Ну и, наверное, самый главный вопрос, где это имеет смысл применять? Давайте представим, что мы делаем игру «Крестики-нолики» и хотели бы на каждом шаге определять, есть ли выигрышная позиция, например, у крестиков? Для простоты сделаем это следующим образом. Все поле из девяти клеток я представлю одномерным списком (сейчас вы узнаете, зачем):

```
P = ['x', 'x', 'o', 'x', 'x', 'x', 'o', 'x', 'x']
```



Тогда, чтобы проверить выигрышные ситуации по строкам, можно воспользоваться функцией all(), следующим образом:

```
res1 = all(x == 'x' for x in P[:3])
res2 = all(x == 'x' for x in P[3:6])
res3 = all(x == 'x' for x in P[6:])

print(res1, res2, res3)
```

Смотрите, мы здесь используем вложенный вызов функции map(), чтобы правильно преобразовать крестики в True, а нолики – в False, иначе бы все преобразовалось в True. Далее, срез для каждой строки на выходе функции map() обрабатывается функцией all() и получаем результат: True – есть выигрышная комбинация; False – нет выигрышной комбинации.

Кстати, внимательный и грамотный зритель здесь сразу должен возмутиться из-за дублирования кода – три раза записана одна и та же анонимная функция.

Для второй функции я приведу такой короткий пример. Предположим, мы делаем игру «Сапер» и игровое поле также представлено в виде одномерного списка длиной NxN элементов:

```
N = 10
P = [0] * (N*N)
```

Далее, если в этом списке появляется хотя бы одна мина (отметим ее звездочкой):

```
P[4] = '*'
```

то игрок проигрывает. Чтобы узнать, наступил ли игрок на мину, удобно воспользоваться функцией any():

```
loss = any(map(lambda x: x == '*', P))
print(loss)
```

После запуска программы увидим значение True, то есть, игрок проиграл. А если мину поставить в комментарий и снова запустить, то увидим значение False.

Вот два простых примера, где удобно применять эти две функции any() и all().

Кортежи в питоне представляют собой простую структуру данных для группировки произвольных объектов. Кортежи являются неизменяемыми — они не могут быть изменены после их создания.

Обратная сторона кортежей — это то, что мы можем получать данные из них используя только числовые индексы. Вы не можете дать имена отдельным элементам сохранённым в кортеже. Это может повлиять на читаемость кода

Именованные кортежи

Мощная особенность и настоящая гордость языка.

Именованный кортеж (или named tuple) позволяет программисту обращаться к элементу кортежа не по индексу, а через удобочитаемый заранее заданный идентификатор.

Посмотрим на примере:

```
from collections import namedtuple

citizen = namedtuple("гражданин", "name age status")
Ivan = citizen(name='Иван Петров', age=27, status='машинист')
print(Ivan)
print(Ivan.status)
```

Точечная нотация при обращении к свойству объекта может вызвать невольную ассоциацию с классами. В общем-то одно из применений namedtuple как раз связано с ситуациями, когда нужно передать несколько свойств объекта одним куском.

Это одна из прекрасных фич в питоне, которая скрыта с первого взгляда.

Гораздо чаще используются словари.

Введение в словари (dict). Базовые операции над словарями

Итак мы уже изучили все перебираемые по индексу типы данных

```
s = 'Python'  # строка (str)
l = [3, 7, 5.5, 'Python', True]  # список  (list)
t = (3, 7, 5.5, 'Python', True) # множество  (tuple)
r = range(20) # прогрессия (range)
```

Это строка и 3 коллекции. Доступ к элементам через квадратные скобки по индексу.

Но представим, что мы должны шарики раскладывать по цветам, каждый в свою коробочку.

```
# red
# blue
# yellow
# green
```

Красные в нулевую, синие во вторую, желтые в третью и т.д.
Но как только шариков станет много 7±2 человеческая память начнет давать сбой.
Мы начнем забывать под каким индексом где-какие шарики лежат.

Т.е. если цветов станет 20, мы будем путаться с коробочками.
А если коробочки подписать, то всё встанет на место. Человек проще с именами.

```
ball_lst = [5,7,11,10]
```

Поэтому во всех языках программирования присутствуют ассоциативные списки, хеш-таблицы (ключ-значение), мапы, dict.

Словарь в Python (он же ассоциативный список) — это тот-же список, но с небольшими отличиями.

Индексов нет. Точнее, в качестве индекса (который в словаре будет называться ключ) могут выступать не только числа, но и любые другие типы данных (даже другие коллекции!).

Грубо говоря, имена и значения.

А вместо индексов – ключи!

```
balls = {'red': 5, 'blue': (1, 2, 3), 'green': [10, 11]}
print(balls)
```

Н уровне структуры данных ключи словаря – это неповторяющиеся значения.

Ключи уникальны, а значения могут повторяться.

Таким образом, ключи это? Неизменяемые типы данных! Какие?

str, int, float, bool, tuple, range

Значение словаря? Любые типы данных!

С точки зрения

Словарь изменяемый тип данных.

Когда мы в памяти переменные создаем. Где они хранятся?

С словаре! Имя переменной и есть ключ. Гарантированно!

Словарь легко преобразуется в json. Раньше был xml, но пик популярности у него прошел.

Обращение к значениям словаря происходит по ключам

```
print(balls['blue'])
```

Если ключа нет, мы получим ошибку

Где коллизия?

```
balls = {'red': 5, 'blue': (1, 2, 3), 'blue': [10, 11]}
print(balls)
print(balls['blue'])
```

Какое значение получим по ключу blue?

А сейчас?

```
balls = {
    'red': 5,
    'blue': (1, 2, 3),
    'blue': [10, 11],
    'blue': {'red': 5, 'blue': (1, 2, 3), 'blue': [10, 11]}
}
print(balls)
print(balls['blue'])
```

А в такой ситуации

```
print(balls)
print(balls['blue']['blue'])
print(balls['blue'])
```

Также допустимы пропуски, если мы все-таки будем использовать в качестве ключа целое число, например у нас может быть элемент связанный с ключем 5, но при этом отсутствовать элемент связанный с ключем 4.

Что все это значит на практике? Всего-лишь, то, что в квадратных скобках для обращения к элементу по “индексу” мы можем указывать произвольный тип, например allMyCats[“Murka”].

Давайте поэкспериментируем с шарами. С учетом знаний типов данных

```
balls = {
    'red': 5,
    'blue': (1, 2, 3),
    'blue': [10, 11],
    'blue': {'red': 5, 'blue': (1, 2, 3), 'blue': [10, 11]}
}
```

Что мы увидим в консоли

```
balls['red'] += 5
balls['blue'] += 5
balls['blue']['red'] += 5
balls['black'] = 10

print(balls)
```

Далее

Вернемся в коробочка с шарами

```
balls = {
    'red': 5,
    'blue': 1,
    'green': 2,
    'yellow': 0
}
```

Поищем синие шары

```
print ('blue' in balls)
print(balls.get('pink', 'Нет такого ключа'))
```

Следующая ситуация

```
if 'white' not in balls:
    balls['white'] = 1
else:
    balls['white'] = 1

print(balls)

pprint.pprint(balls, width=20)
```

Без знания физики мир становится удивительным и чудесным.

Без знания химии, человек верит в любовь, а со знанием биохимии гормонов строит долгий и счастливый брак.

Так же и в программировании. Чем дальше, тем меньше магии и волшебства.

Одни и те же механизмы.

Практическая задача: Найти частоту вхождения каждого символа в строку

‘Hello, Python’

```
str_ = 'Hello, Python'.replace(' ','')

count_char = {}

for char in str_:
    if char in count_char:
        count_char[char] += 1
    else:
        count_char[char] = 1


pprint.pprint(count_char, width=20)
```

Еще короче

```
for char in str_:
    if not char in count_char:
        count_char[char] = str_.count(char)
```

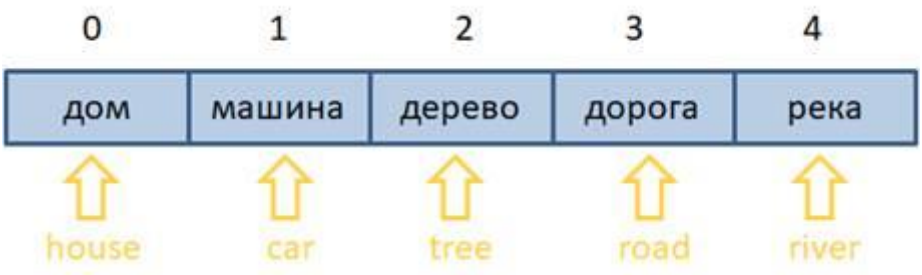
или `if char not in count_char:`

Давайте представим, что мы в программе хотели бы описать, следующие зависимости:

house -> дом
car -> машина
tree -> дерево
road -> дорога
river -> река

Если определить значения через список:

```
d = ["дом", "машина", "дерево", "дорога", "река"]
```



то мы получим коллекцию, где каждое значение (русское слово) будет связано с числом – индексом. И изменить эти индексы невозможно – они создаются автоматически самим списком. А нам бы хотелось обращаться к этим значениям

по английским словам. Как раз это позволяет делать словарь. Он формирует коллекцию в виде ассоциативного массива с доступом к элементу по ключу.

Для создания словаря используется следующий синтаксис:

{key1: value1, key2: value2, …, keyN:valueN}

и он определяет неупорядоченную коллекцию данных, то есть, данные в словаре не имеют строгого порядка следования. Располагаться они могут произвольным образом, но всегда связаны с одним, строго определенным ключом.

В нашем конкретном случае словарь можно определить, так:

```
d = {"house": "дом", "car": "машина",
    "tree": "дерево", "road": "дорога",
    "river": "река"}
```

Как видите, мы можем записывать значения в несколько строчек, не обязательно все писать в одну.

После создания словаря, мы можем по ключу получать нужное нам значение. Для этого записываются квадратные скобки и в них указывается ключ:

```
d["house"]
```

возвращается значение «дом», которое связано с этим ключом. Если же указать не существующий ключ:

```
d[100]
```

то получим ошибку. Разумеется, ключи в словарях всегда уникальны. Если записать два одинаковых:

```
d = {"house": "дом", "house": "дом 2", "car": "машина"}
```

то ключ «house» будет ассоциирован с последним значением – «дом 2».

Также для определения словаря в Python существует специальная встроенная функция dict(). Ей, в качестве аргументов, через запятую перечисляются пары в формате ключ=значение:

```
d2 = dict(one = 1, two = 2, three = "3", four = "4")
```

Здесь ключи преобразовываются в строки и должны определяться, как и имена переменных. Например, использовать числа уже не получится:

```
d2 = dict(1 = "one", two = 2, three = "3", four = "4")
```

Это неверное имя переменной. Обратите внимание на эти два способа создания словарей.

Один из плюсов этой функции – возможность создать словарь из списка специального вида. Что это за вид? Например, такой:

```
lst = [[2, "неудовлетворительно"], [3, "удовлетворительно"], [4, "хорошо"], [5, "отлично"]]
```

Здесь вложенные списки состоят из двух элементов, которые функцией dict() интерпретируются как ключ и значение. Если мы сформируем словарь:

```
d3 = dict(lst)
```

то, как раз, увидим такую структуру данных. Причем, в этом случае, в качестве ключей можно уже выбирать и другие типы данных, не только строки, например, числа.

Если вызвать эту функцию без параметров:

```
d = dict()
```

то получим пустой словарь. Это эквивалентно такой записи:

```
d = {}
```

и, чаще всего, она используется на практике, так как короче.

Давайте теперь ответим на вопрос: а что вообще можно использовать в качестве ключей? Какие типы данных? В наших примерах это было или число, или строка. Что можно брать еще? На самом деле любые неизменяемые типы. Обратите внимание – неизменяемые! Например, можно написать так:

```
d[True] = "Истина"
d[False] = "Ложь"
```

Здесь ключи – булевы значения. В результате, получим словарь:

{True: 'Истина', False: 'Ложь'}

Также этот пример показывает, что, присваивая словарю значение с новым ключом, оно автоматически добавляется в словарь. В результате, наш изначально пустой словарь стал содержать две записи. Если же мы существующему ключу присваиваем другое значение:

```
d[True] = 1
```

то он просто поменяет свое значение в словаре. Вот так можно добавлять и менять значения словаря. То есть, словарь относится к изменяемым типам.

А вот если ключом указать список:

```
d[[1,2]] = 1
```

то возникнет ошибка, т.к. список – это изменяемый объект. Вообще, чаще всего в качестве ключей используются строки или числа.

Это то, что касается ключей, а вот на значения словаря никаких ограничений не накладывается – там могут самые разные данные:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
```

Операторы и функции работы со словарем

В заключение этого занятия рассмотрим некоторые операторы и функции работы со словарем. С помощью функции:

```
len(d)
```

можно определить число элементов в словаре. Оператор:

```
del d[True]
```

выполняет удаление пары ключ=значение для указанного ключа. Если записать несуществующий ключ:

```
del d["abc"]
```

то оператор возвращает ошибку. Поэтому перед удалением лучше проверить существует ли данный ключ в словаре с помощью оператора in:

```
"abc" in d
```

Обратите внимание, оператор in проверяет именно наличие ключа, а не значения. Например, если добавить это значение:

```
d[1] = "abc"
```

и повторно выполнить команду:

```
"abc" in d
```

то увидим значение False.

Также можно делать противоположную проверку на отсутствие ключа, прописывая дополнительно оператор not:

```
"abc" not in d
```

Надеюсь, вы поняли, что такое словарь, как можно его задавать и обращаться к элементам этой коллекции по ключам. Познакомились с функцией dict() для создания словарей, функцией len() – для определения числа элементов и операторами in и del. Для закрепления этого материала подготовлены практические задания, после которых мы продолжим работу со словарями.

Задания на словари-1

Задание 1. Вводятся данные в формате ключ=значение в одну строчку через пробел. Значениями здесь являются целые числа (см. пример ниже). Необходимо на их основе создать словарь d с помощью функции dict() и вывести его на экран командой:

```
print(*(d.items()))
```

Входные данные:

```
one=1 two=2 three=3
```

Выходные данные:

```
('one', 1) ('three', 3) ('two', 2)
```

```
s = list((input().split(' ')))
d = {}
for i in s:
    d[i.split('=')[0]] = int(i.split('=')[1])
print (*sorted(d.items()))
```

```
lst = [i.split('=') for i in input().split()]
d = {i: int(v) for i, v in lst}
print(*sorted(d.items()))
```

Задание 2. На вход программы поступают данные в виде набора строк в формате:

```
ключ1=значение1
ключ2=значение2
...
ключN=значениеN
```

Ключами здесь выступают целые числа (см. пример ниже). Необходимо их преобразовать в словарь d (без использования функции dict()) и вывести его на экран командой:

`print(*sorted(d.items()))`

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Входные данные:

5=отлично

4=хорошо

3=удовлетворительно

Выходные данные:

(3, 'удовлетворительно') (4, 'хорошо') (5, 'отлично')

```
lst_in = ["5=отлично", "4=хорошо", "3=удовлетворительно"]

d = {}

for i in lst_in:
    x,y = i.split('=')
    d[int(x)]=y

print (*sorted(d.items()))
```

или

```
lst = [i.split('=') for i in lst_in]

d = {int(i): v for i, v in lst}

print(*sorted(d.items()))
```

или

```
d = {}

for s in lst_in:

    row = s.split('=')

    d[int(row[0])] = row[1]
```

```
print(*sorted(d.items()))
```

Задание 3. Вводятся данные в формате ключ=значение в одну строчку через пробел. Необходимо на их основе создать словарь, затем проверить, существуют ли в нем ключи со значениями: 'house', 'True' и '5' (все ключи - строки). Если все они существуют, то вывести на экран ДА, иначе - НЕТ.

Входные данные:

```
вологда=город house=дом True=1 5=отлично 9=божественно
```

Выходные данные:

```
ДА
```

```
s0 = 'вологда=город house=дом True=1 5=отлично 9=божественно'
s1 = list(map(str, s0.split(' ')))
d = {}
for s in s1:
    key,value = s.split('=')
    d[key]=value


if 'house' in d and 'True' in d and '5' in d:
    print ('ДА')

else:
    print('НЕТ')
```

или

```
d = dict([i.split('=') for i in input().split()])

print('ДА' if 'house' in d and 'True' in d and '5' in d else 'НЕТ'


print('ДА' if all(i in d for i in ('house', 'True', '5')) else "НЕТ")
```

Методы словаря, перебор элементов словаря в цикле

Мы продолжаем изучение словарей языка Python. Это занятие начнем с ознакомления основных методов, которые есть у этой коллекции.

Посмотрим на

```
d = {'one': '1', 'two': '2', 'three': '3'}
print(d.values())
print(d.keys())
print(d.items())
```

Результат:

```
dict_values(['1', '2', '3'])

dict_keys(['one', 'two', 'three'])

dict_items([('one', '1'), ('two', '2'), ('three', '3')])
```

Метод items возвращает множество-подобный (set-like), состоящий из уникальных не повторяющихся элементов, объект типа dict_items, состоящий из списка кортежей пар ключ-значение.

Метод keys возвращает множество-подобный (set-like), состоящий из уникальных не повторяющихся элементов, объект типа dict_keys, состоящий из списка ключей словаря.

Метод values возвращает множество-подобный (set-like), состоящий из уникальных не повторяющихся элементов, объект типа dict_values, состоящий из списка значений словаря.

```
d = {'one': '1', 'two': '2', 'three': '3'}
it = d.items()
print(it)
del d['one']
print(it)
```

Результат:

```
dict_items([('one', '1'), ('two', '2'), ('three', '3')])

dict_items([('two', '2'), ('three', '3')])6
```

Соответственно, мы можем и дальше преобразовывать эти структуры данных:

```
d = {'one': '1', 'two': '2', 'three': '3'}
vals = d.values()

print(list(vals)[0)
```

и так далее

```
vals = d.values()
keys = d.keys()
```

```
items = d.items()

print(list(vals))
print(list(keys))
print(list(items))
```

```
['1', '2', '3']

['one', 'two', 'three']

[('one', '1'), ('two', '2'), ('three', '3')]
```

Т.е. изначально работать с такими set-подобными структурами мы не можем, а после преобразования – легко!

И даже так

```
print(dict(items))
```

получим исходный словарь

```
{'one': '1', 'two': '2', 'three': '3'}
```

Начнем с метода

dict.fromkeys(список[, значение по умолчанию])

который формирует словарь с ключами, указанными в списке и некоторым значением. Например, передадим методу список с кодами стран:

```
a = dict.fromkeys(["+7", "+6", "+5", "+4"])
```

в результате, получим следующий словарь:

```
{'+7': None, '+6': None, '+5': None, '+4': None}
```

Обратите внимание, все значения у ключей равны None. Это значение по умолчанию. Чтобы его изменить используется второй необязательный аргумент, например:

```
a = dict.fromkeys(["+7", "+6", "+5", "+4"], "код страны")
```

на выходе получаем словарь с этим новым указанным значением:

```
{'+7': 'код страны', '+6': 'код страны', '+5': 'код страны', '+4': 'код страны'}
```

Преобразуем список в словарь при помощи генератора словаря

Для преобразования списка Python в словарь также можно использовать генератор словаря.

Генератор словаря похож на генератор списка в том, что оба они создают новое значение соответствующего типа данных.

```
fruits = ["Apple", "Pear", "Peach", "Banana"]
fruit_dictionary = {fruit : "In stock" for fruit in fruits}
print(fruit_dictionary
```

Для начала мы объявили список фруктов (fruits), где хранятся их названия, которые мы хотим перенести в словарь.

Затем мы использовали генератор словаря, чтобы пройтись по каждому элементу в списке fruits. Для каждого фрукта в нашем списке мы добавили элемент в новый словарь. При этом каждому фрукту мы присвоили значение In stock.

Наконец, мы вывели наш новый словарь в консоль. Можем запустить наш код, и он выведет то же самое, что и в первом примере.

Конвертируем два списка в словарь при помощи zip()

В нашем последнем примере мы преобразовываем в словарь один список и присваиваем одинаковое значение по умолчанию для каждого ключа в словаре. Но можно преобразовать в словарь и два списка. Давайте попробуем это сделать.

Для этого мы можем использовать такую функцию Python, как zip(). Она позволяет объединить два списка. Элементы одного списка мы можем использовать в качестве ключей для словаря, а элементы второго — в качестве значений.

Итак, предположим, что у нас есть два списка: один содержит названия фруктов, а другой – их цены. Мы хотим создать единый словарь, в котором будет храниться название фрукта и его цена.

Для выполнения этой задачи мы можем использовать следующий код:

```
fruits = ["Apple", "Pear", "Peach", "Banana"]

prices = [0.35, 0.40, 0.40, 0.28]

fruit_dictionary = dict(zip(fruits, prices))

print(fruit_dictionary)
```

Что же мы сделали? Во-первых, мы определили два списка: список фруктов fruits и список цен prices.

Затем мы использовали функцию zip(), чтобы объединить наши списки. Эта функция возвращает список кортежей. Поскольку нам нужен словарь, для преобразования наших кортежей мы использовали dict().

В конце мы вывели содержимое нашего нового словаря в консоль.

Проверим себя

```
months = 'Сентябрь', 'Октябрь', 'Ноябрь', 'Сентябрь'
days = 1, 2, 3

res = dict(zip(months,days))

print(len(res))
```

Какая длина словаря

Результат

{'Сентябрь': 1, 'Октябрь': 2, 'Ноябрь': 3}

А если вот так

```
days = 1, 2, 3, 4
```

А сейчас какая длина у словаря

```
months = 'Сентябрь', 'Октябрь', 'Сентябрь', 'Ноябрь', 'Декабрь'
days = 1, 2, 3, 4

res = dict(zip(months,days))
```

В случае с кортежами или списками, ситуация была бы другая.

Следующий метод

```
d.clear()
```

служит для очистки словаря, то есть, удаления всех его элементов.

Далее, для создания копии словаря используется метод copy:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
d2 = d.copy()
d2["list"] = [5,6,7]
print(d)
print(d2)
```

Также копию можно создавать и с помощью функции dict(), о которой мы с вами говорили на предыдущем занятии:

```
d2 = dict(d)
```

Результат будет абсолютно таким же, что и при вызове метода copy().

Следующий метод get позволяет получать значение словаря по ключу:

```
d.get("list")
```

Его отличие от оператора

```
d["list"]
```

в том, что при указании неверного ключа не возникает ошибки, а выдается по умолчанию значение None:

```
print(d.get(3))
```

Это значение можно изменить, указав его вторым аргументом:

```
print( d.get(3, False) )
```

Похожий метод

dict.setdefault(key[, default])

возвращает значение, ассоциированное с ключом key и если его нет, то добавляет в словарь со значением None, либо default – если оно указано:

```
d.setdefault("list")
d.setdefault(3)
```

Добавит ключ 3 со значением None. Удалим его:

```
del d[3]
d.setdefault(3, "three")
```

тогда добавится этот ключ со значением «three». То есть, этот метод способен создать новую запись, но только в том случае, если ключ отсутствует в словаре.

Следующий метод

```
d.pop(3)
```

удаляет указанный ключ и возвращает его значение. Если в нем указывается несуществующий ключ, то возникает ошибка:

```
d.pop("abc")
```

Но мы можем легко исправить ситуацию, если в качестве второго аргумента указать значение, возвращаемое при отсутствии ключа:

```
d.pop("abc", False)
```

Здесь возвратится False. Если же ключ присутствует, то возвращается его значение.

Следующий метод

```
d.popitem()
```

выполняет удаление произвольной записи из словаря. Если словарь пуст, то возникает ошибка:

```
d2 = {}
d2.popitem()
```

Следующий метод

```
d.keys()
```

возвращает коллекцию ключей. По умолчанию цикл for обходит именно эту коллекцию, при указании словаря:

```
d = {True: 1, False: "Ложь", "list": [1,2,3], 5: 5}
for x in d:
    print(x)
```

то есть, эта запись эквивалента такой:

```
for x in d.keys():
    ...
```

Если же вместо keys записать метод values:

```
for x in d.values():
    ...
```

то обход будет происходить по значениям, то есть, метод values возвращает коллекцию из значений словаря.

Последний подобный метод items

```
for x in d.items():
    ...
```

возвращает записи в виде кортежей: ключ, значение. О кортежах мы уже говорили, здесь лишь отмечу, что к элементу кортежа можно обратиться по индексу и вывести отдельно ключи и значения:

```
print(x[0], x[1])
```

Или, используя синтаксис множественного присваивания:

```
x,y = (1, 2)
```

можно записать цикл for в таком виде:

```
for key, value in d.items():
    print(key, value)
```

что гораздо удобнее и нагляднее.

Следующий метод update() позволяет обновлять словарь содержимым другого словаря. Например, есть два словаря:

```
d = dict(one = 1, two = 2, three = "3", four = "4")
d2 = {2: "неудовлетворительно", 3: "удовлетворительно", 'four': "хорошо", 5: "отлично"}
```

И мы хотим первый обновить содержимым второго:

```
d.update(d2)
```

Смотрите, ключ four был заменен строкой «хорошо», а остальные, не существующие ключи были добавлены.

Еще способ, доступный в Pt с версии 3.9

```
d1 = {'Сентябрь': 10, 'Октябрь': 11, 'Ноябрь': 12}
d2 = {'Январь': 1, 'Февраль': 2, 'Март ': 3, 'Сентябрь': 100}

d3 = d1 | d2
d1.update(d2)

print(d3)
print(d1)
```

Или

Если же нам нужно создать новый словарь, который бы объединял содержимое обоих, то для этого можно воспользоваться конструкцией:

```
d3 = {**d1, **d2}
```

Однако, детально как она работает будет понятно позже, когда мы познакомимся с операторами распаковки коллекций.

```
holidays = {
    'January': (1, 2, 3, 4, 5, 6, 7, 8),
    'February': (23),
    'March': (8)
}
```

Что мы увидим в принте?

```
print(holidays['January'][0])
```

А в этом случае

```
holidays = {
    1: (1, 2, 3, 4, 5, 6, 7, 8),
    2: (23),
    3: (8)
}

print(holidays[1][0])
```

А сейчас как вытащить цифру 1

```
holidays = {
    (1,2) : (1, 2, 3, 4, 5, 6, 7, 8),
    2: (23),
    3: (8)
}
```

Итак, на этом занятии мы с вами познакомились с основными методами словарей, а также способами их перебора и объединения. Для закрепления этого материала выполните практические задания и переходите к следующему уроку.

Задания на словари-2

Задание 1. Вводится список целых чисел в одну строчку через пробел. С помощью словаря выделите только уникальные (не повторяющиеся) введенные значения и, затем, сформируйте список из уникальных чисел. Выведите его на экран в виде набора чисел, записанных через пробел. Решить двумя способами (через множества и словари)

Входные данные:

```
8 11 -4 5 2 11 4 8
```

Выходные данные:

```
8 11 -4 5 2 4
```

```
d = {}

s =  list(map(int, input().split()))

for i in s:

    if i in d:

        continue

    else:
```

```
    d[i] = i

print (*d.keys())
```

Задание 2. Вводятся данные в формате ключ=значение в одну строчку через пробел. Необходимо на их основе создать словарь d, затем удалить из этого словаря ключи 'False' и '3', если они существуют. Ключами и значениями словаря являются строки. Вывести полученный словарь на экран командой:

```
print(*sorted(d.items()))
```

Входные данные:

лена=имя дон=река москва=город False=ложь 3=удовлетворительно True=истина

Выходные данные:

('True', 'истина') ('дон', 'река') ('лена', 'имя') ('москва', 'город')

```
s = input().split() # ввод данных 'one=1','two=2','three=3'
lst = [s[i].split('=') for i in range(len(s))] # разбивка данных .,через list comprehension [s[элемент].split('=') пробегаемся по длинне списка (всего 3 элемента)].Таким образом у нас сейчас данные получаются 'one' '1' и т.д.
d = dict(lst) # делаем словарь
for key in d: # цикл чтобы получить пару строка : число
d[key] = int(d[key])
print(*sorted(d.items()))
```

Задание 3. Вводятся номера телефонов в одну строчку через пробел с разными кодами стран: +7, +6, +2, +4 и т.д. Необходимо составить словарь d, где ключи - это коды +7, +6, +2 и т.п., а значения - список номеров (следующих в том же порядке, что и во входной строке) с соответствующими кодами. Полученный словарь вывести командой:

```
print(*sorted(d.items()))
```

Входные данные:

+71234567890 +71234567854 +61234576890 +52134567890 +21235777890 +21234567110 +71232267890

Выходные данные:

('+2', ['+21235777890', '+21234567110']) ('+5', ['+52134567890']) ('+6', ['+61234576890']) ('+7', ['+71234567890', '+71234567854', '+71232267890'])

```
lst = input().split() # создаём список номеров
d_keys = dict([[number[:2], 0] for number in lst]) # создаём словарь, состоящий из кодов операторов в качестве клюей и "пустых значений". (Поскольку нам от этого словаря нужны только ключи, лучше бы мы использовали set(), но, пока мы не знаем, что такое set(), такой костыль)
d = dict([[key, [number for number in lst if key in number]] for key in d_keys]) # по найденым ключам формируем требуемый словарь.
print(*sorted(d.items())
```

Задание 4. Пользователь вводит в цикле целые положительные числа, пока не введет число 0. Для каждого числа вычисляется квадратный корень (с точностью до сотых) и значение выводится на экран (в столбик). С помощью словаря выполните кэширование данных так, чтобы при повторном вводе того же самого числа результат не вычислялся, а бралось ранее вычисленное значение из словаря. При этом на экране должно выводиться:

значение из кэша: <число>

Входные данные:

1
2
3
3
2
4
0

Выходные данные:

1.0
1.41
1.73
значение из кэша: 1.73
значение из кэша: 1.41
2.0

```
d = {}
while True:
```

```
x = int(input())
if x == 0:
    break
if x in d:
    print ('значение из кэша:', d[x])
else:
    d[x] = round(x ** 0.5,2)
    print (d[x])
```

Задание 5. Тестовый веб-сервер возвращает HTML-страницы по URL-адресам (строкам). На вход программы поступают различные URL-адреса. Если адрес пришел впервые, то на экране отобразить строку (без кавычек):

"HTML-страница для адреса <URL-адрес>"

Если адрес **приходит** повторно, то следует взять строку "HTML-страница для адреса <URL-адрес>" из словаря и вывести на экран сообщение (без кавычек):

"Взято из кэша: HTML-страница для адреса <URL-адрес>"

Сообщения выводить каждое с новой строки.

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Входные данные:

```
ustanovka-i-zapusk-yazyka
ustanovka-i-poryadok-raboty-pycharm
peremennyye-operator-prisvaivaniya-tipy-dannykh
arifmeticheskiye-operatsii
ustanovka-i-poryadok-raboty-pycharm
```

Выходные данные:

```
HTML-страница для адреса ustanovka-i-zapusk-yazyka
HTML-страница для адреса ustanovka-i-poryadok-raboty-pycharm
HTML-страница для адреса peremennyye-operator-prisvaivaniya-tipy-dannykh
HTML-страница для адреса arifmeticheskiye-operatsii
Взято из кэша: HTML-страница для адреса ustanovka-i-poryadok-raboty-pycharm
```

Задание 6. Имеется словарь с наименованиями предметов и их весом (в граммах):

```
things = {'карандаш': 20, 'зеркальце': 100, 'зонт': 500, 'рубашка': 300,
          'брюки': 1000, 'бумага': 200, 'молоток': 600, 'пила': 400, 'удочка': 1200,
          'расческа': 40, 'котелок': 820, 'палатка': 5240, 'брезент': 2130, 'спички': 10}
```

Сергей собирается в поход и готов взвалить на свои хрупкие плечи максимальный вес в N кг (вводится с клавиатуры). Он решил класть в рюкзак предметы в порядке убывания их веса (сначала самые тяжелые, затем, все более легкие) так, чтобы их суммарный вес не превысил значения N кг. Все предметы даны в единственном экземпляре. Выведите список предметов (в строчку через пробел), которые берет с собой Сергей в порядке убывания их веса.

P. S. 1 кг = 1000 грамм

Входные данные:

10

Выходные данные:

палатка брезент удочка брюки пила карандаш спички

```
things = {'карандаш': 20, 'зеркальце': 100, 'зонт': 500, 'рубашка': 300,
'брюки': 1000, 'бумага': 200, 'молоток': 600, 'пила': 400, 'удочка': 1200,
'расческа': 40, 'котелок': 820, 'палатка': 5240, 'брезент': 2130, 'спички': 10}

n = int(input()) * 1000
res = 0
for key in sorted(things, key = lambda x : things.get(x), reverse = True):
    weight = things.get(key)
    if res + weight > n:
        continue
    else:
        res += weight
        print(key,end = ' ')
```

Множества (set) и их методы

На этом занятии мы с вами познакомимся с последней базовой коллекцией – множествами.

Формально, множество (set) – это неупорядоченная коллекция уникальных элементов. Уникальных, то есть, в ней отсутствуют дублирующие значения.

Для задания множества используются фигурные скобки, в которых через запятую перечисляются значения элементов:

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

Это немного похоже на определение словаря, но в отличие от него, здесь не прописываются ключи – только значения. В результате получаем совершенно другой тип объекта – множество:

```
type(a)
```

Так как множество состоит только из уникальных значений, то попытка в него поместить несколько одинаковых значений:

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

приведет к тому, что останутся только не повторяющиеся. Это ключевая особенность работы данной коллекции – она автоматически отбрасывает все дубли.

Во множество мы можем записывать только неизменяемые данные, такие как:

- числа;
- булевы значения;
- строки;
- кортежи

```
a = {1, 4.5, "hi", "hi", (4, True), ("a", False)}
```

А вот изменяемые типы данных:

- списки;
- словари;
- множества

добавлять нельзя, будет ошибка:

```
b = {[1, 2], 3, 4}
b = {{1: 1, 2: 2}, 3, 4}
b = {{1, 2}, 3, 4}
```

Для создания множества используется встроенная функция set(). Если ее записать без аргументов:

```
set()
```

то получим пустое множество. Но, обратите внимание, если попытаться создать пустое множество вот так:

```
b = {}  # словарь
```

то получим не пустое множество, а пустой словарь! Пустое множество создается именно с помощью функции set(). Вот этот момент нужно запомнить.

Если указать какой-либо итерируемый объект, например, список:

```
set([1, 2, 1, 0, -1, 2])
```

то получим множество, состоящее из уникальных значений этого списка. Или, если передать строку:

```
set("abrakadabra")
```

то увидим множество из уникальных символов этой строки. И так для любого итерируемого объекта, даже можно прописать функцию range():

```
b = set(range(7))
```

Обратите внимание, множества представляют собой неупорядоченную коллекцию, то есть, значения элементов могут располагаться в них в любом порядке. Это также означает, что с множествами нельзя выполнять операции доступа по индексу или срезам:

```
b[0]
```

приведет к ошибке.

Для чего вообще может потребоваться такая коллекция в программировании? Я, думаю, большинство из вас уже догадались – с их помощью, например, можно легко и быстро убирать дубли из данных. Представим, что некоторая логистическая фирма составила список городов, куда доставляли товары:

```
cities = ["Калуга", "Краснодар", "Тюмень", "Ульяновск", "Москва", "Тюмень", "Калуга", "Ульяновск"]
```

И отдел статистики хотел бы получить список уникальных городов, с которыми было сотрудничество. Очевидно, это можно сделать с помощью множества:

```
set(cities)
```

Обратите внимание, порядок городов может быть любым, так как множество – неупорядоченная коллекция данных. Мало того, мы можем этот полученный перечень снова преобразовать в список, используя функцию:

```
list(set(cities))
```

Все, мы отобрали уникальные города и представили их в виде списка. Видите, как это легко и просто можно сделать с помощью множеств.

А что если нам не нужен список уникальных городов, а достаточно лишь перебрать все элементы полученного множества:

```
q = set(cities)
```

Учитывая, что множество – это итерируемый объект, то пройтись по всем его элементам можно с помощью цикла for:

```
for c in q:
    print(c)
```

Здесь переменная с последовательно ссылается на значения элементов множества q и соответствующий город выводится в консоль. Или, то же самое можно сделать через механизм итераторов:

```
it = iter(q)
next(it)
```

Наконец, функция:

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

возвратит нам число элементов в множестве. А для проверки наличия значения в множестве можно воспользоваться, уже знакомым нам оператором in:

```
"abc" in a
```

Он возвращает True, если значение имеется и False в противном случае. Или можно проверить на непринадлежность какого-либо значения:

```
7 not in a
```

Методы добавления/удаления элементов множества

В заключение этого занятия рассмотрим методы для добавления и удаления элементов в множествах. Первый метод add() позволяет добавлять новое значение в множество:

```
b.add(7)
```

Значение будет добавлено только в том случае, если оно отсутствует в множестве b. Если попробовать, например, еще раз добавить семерку:

```
b.add(7)
```

то множество не изменится, но и ошибок никаких не будет.

Если в множество требуется добавить сразу несколько значений, то для этого хорошо подходит метод update():

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

В качестве аргумента указывается любой итерируемый объект, в данном случае, список, значения которого добавляются в множество с проверкой уникальности. Или, можно передать строку:

```
b.update("abrakadabra")
```

Получим добавку в виде уникальных символов этой строки. И так далее, в качестве аргумента метода update() можно указывать любой перебираемый объект.

Далее, для удаления элемента по значению используется метод discard():

```
b.discard(2)
```

Если еще раз попытаться удалить двойку:

```
b.discard(2)
```

то ничего не произойдет и множество не изменится.

Другой метод для удаления элемента по значению – remove():

```
b.remove(4)
```

но при повторном таком вызове:

```
b.remove(4)
```

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 неупорядоченная коллекция. 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()
```

На этом мы с вами завершим первое знакомство с множествами. Для закрепления материала, как всегда, вас ждут практические задания, а я буду всех вас ждать на следующем уроке.

Задания на множества - 1

Задание 2. Вводится текст в одну строку, слова разделены пробелом. Необходимо подсчитать число уникальных слов (без учета регистра) в этом тексте. Результат (число уникальных слов) вывести на экран.

Входные данные:

Мама мыла раму а потом мыла кота и еще мыла пол

Выходные данные:

9

Задание 3. 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.

Входные данные:

Python 3.9.11 - best language!

Выходные данные:

1 3 9

Задание 4. В ночном клубе фиксируется список гостей. Причем гости могут выходить из помещения, а затем, снова заходить. Тогда их имена фиксируются повторно. На вход программы поступает такой список (каждое имя записано с новой строки). Требуется подсчитать общее число гостей, которые посетили ночной клуб. Полагается, что гости имеют уникальные имена. На экран вывести общее число гостей клуба.

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Входные данные:

Мария

Елена

Екатерина

Александр

Елена

Мария

Выходные данные:

4

6.4 Множества (set) и их методы

Задание 5. В аккаунте youtube Сергея прокомментировали очередное видео. Некоторые посетители оставляли несколько комментариев. Требуется по списку комментариев определить уникальное число комментаторов. Комментарии поступают на вход программы в формате:

имя 1: комментарий 1
имя 2: комментарий 2

...
имя N: комментарий N

Также полагается, что имена у разных комментаторов не совпадают. Вывести на экран общее число уникальных комментаторов.

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Входные данные:

EvgeniyK: спасибо большое!

LinaTroshka: лайк и подписка!

Sergey Karandeev: крутое видео!

Евгений Соснин: обожаю

EvgeniyK: это повтор?

Sergey Karandeev: нет, это новое видео

Выходные данные:

4

```
lst_in = list(map(str.strip, sys.stdin.readlines()))
print(len(set([i.split(':')[0] for i in lst_in])))
```

Операции над множествами, сравнение множеств

На этом занятии мы с вами будем говорить об операциях над ними – это:

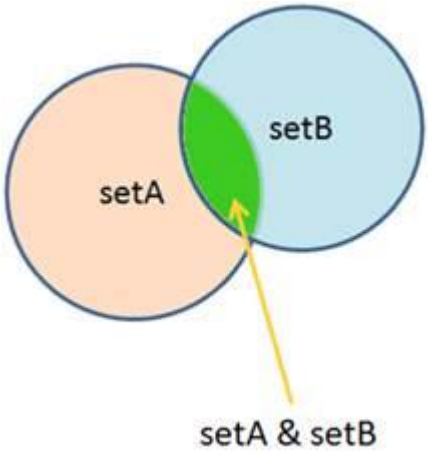
- пересечение множеств;
- объединение множеств;
- вычитание множеств;
- вычисление симметричной разности.

Для простоты зададим два множества с числами:

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

На их основе можно построить третье множество, как результат пересечения этих двух:

```
setA & setB
```



На выходе получаем новое множество, состоящее из значений, общих для множеств setA и setB. Это и есть операция пересечения двух множеств. При этом, исходные множества остаются без изменений. Здесь именно создается новое множество с результатом выполнения этой операции.

Разумеется, чтобы в дальнейшем в программе работать с полученным результатом, его нужно сохранить через какую-либо переменную, например, так:

```
res = setA & setB
```

Также допустима запись в виде:

```
setA &= setB
```

это будет эквивалент строчки:

```
setA = setA & setB
```

То есть, мы вычисляем результат пересечения и сохраняем его в переменной seta и, как бы, меняем само множество setA.

А вот если взять множество:

```
setC = {9, 10, 11}
```

которое не имеет общих значений с другим пересекающимся множеством, то результатом:

```
setA & setC
```

будет пустое множество.

Обычно, на практике используют оператор пересечения &, но вместо него можно использовать специальный метод:

```
setA = {1, 2, 3, 4}
setB = {3, 4, 5, 6, 7}
setA.intersection(setB)
```

Результат будет тем же, на выходе получим третье множество, состоящее из общих значений множеств setA и setB. Если же мы хотим выполнить аналог операции:

```
setA &= setB
```

то для этого следует использовать другой метод:

```
setA.intersection_update(setB)
```

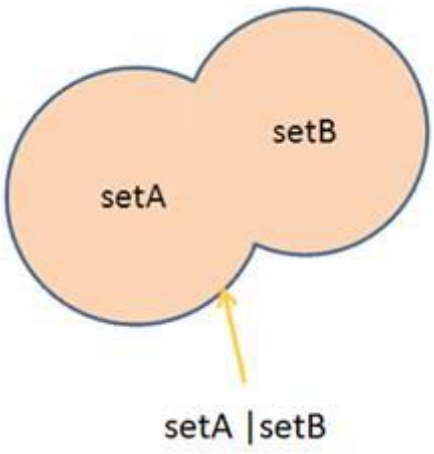
В результате, меняется само множество setA, в котором хранятся значения пересечения двух множеств.

Вот такие способы вычислений пересечения множеств существуют в Python.

Объединение множеств

Следующая операция – это объединение двух множеств. Она записывается, так:

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



На выходе также получаем новое множество, состоящее из всех значений обоих множеств, разумеется, без их дублирования.

Или же можно воспользоваться эквивалентным методом:

```
setA.union(setB)
```

который возвращает множество из объединенных значений. При этом, сами множества остаются без изменений.

Операцию объединения можно записать также в виде:

```
setA |= setB
```

Тогда уже само множество setA будет хранить результат объединения двух множеств, то есть, оно изменится. Множество setB останется без изменений.

Вычитание множеств

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

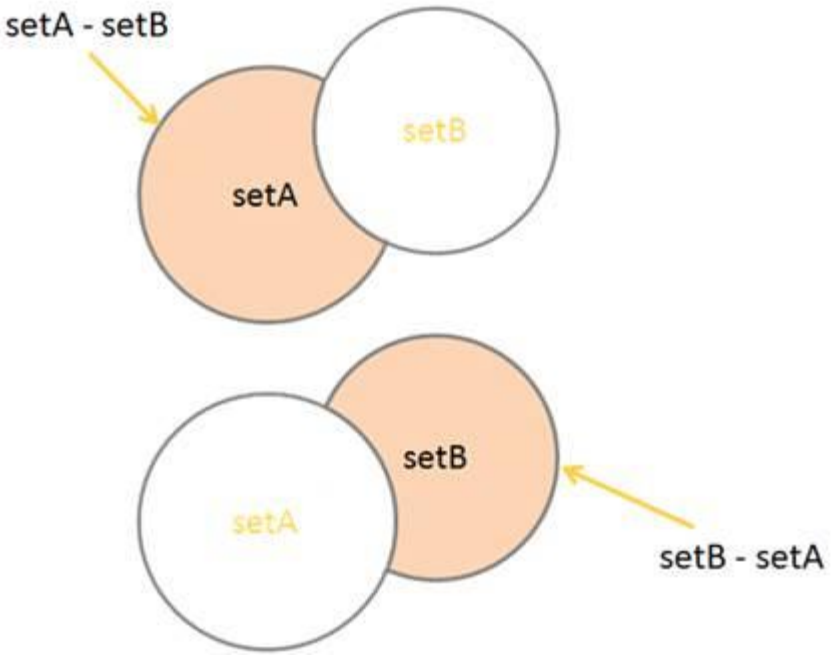
```
setA = {1,2,3,4}
setB = {3,4,5,6,7}
```

операция

```
setA - setB
```

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

{1, 2}



Или, наоборот, если из множества setB вычесть множество setA:

```
setB - setA
```

то получим значения из которых исключены величины, входящие в множество setA.

Также здесь можно выполнять эквивалентные операции:

```
setA -= setB
setB -= setA
```

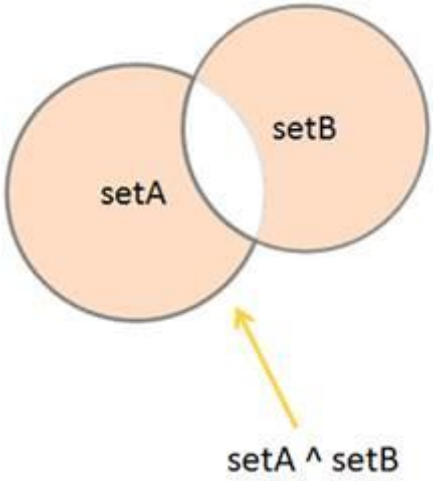
В этом случае переменные setA и setB будут ссылаться на соответствующие результаты вычитаний.

Симметричная разность

Наконец, последняя операции над множествами – симметричная разность, возвращает множество, состоящее из неповторяющихся значений обоих множеств. Реализуется она следующим образом:

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

На выходе получим третье, новое множество, состоящее из значений, не входящих одновременно в оба множества.



Сравнение множеств

В заключение этого занятия я хочу показать вам, как можно сравнивать множества между собой.

Предположим, имеются два таких множества:

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

И мы хотим узнать, равны они или нет. Для этого, используется уже знакомый нам оператор сравнения на равенство:

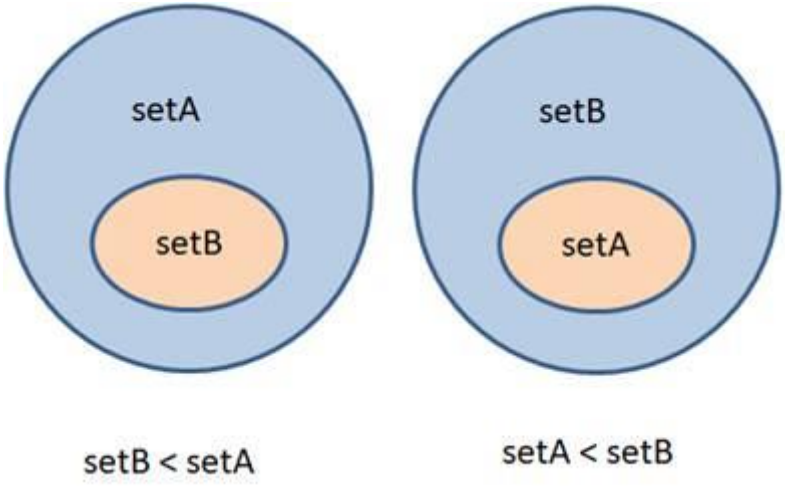
```
setA == setB
```

Увидим значение True (истина). Почему? Дело в том, что множества считаются равными, если все элементы, входящие в одно из них, также принадлежат другому и мощности этих множеств равны (то есть они содержат одинаковое число элементов). Наши множества setA и setB удовлетворяют этим условиям.

Соответственно, противоположное сравнение на не равенство, выполняется, следующим образом:

```
setA != setB
```

Следующие два оператора сравнения на больше и меньше, фактически, определяют, входит ли одно множество в другое или нет.



Например, возьмем множества

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

тогда операция

```
setB < setA
```

вернет True, а операция

```
setB > setA
```

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.

Я, думаю, из этих примеров понятно, как выполняется сравнение множеств между собой.

Для закрепления этого материала, как всегда, пройдите практические задания и, затем, дальше – на покорение новых вершин языка Python. Буду вас ждать на следующем уроке.

Frozenset (неизменяемое множество)

```
a = frozenset('hellow')

b = set('hellow')

print(a == b)

print(type(a - b))

print(type(b | a))

b.add('q')

# a.add('q') # вызовет ошибку
```

Задания на множества - 2

Задание 1. Вводятся два списка целых чисел каждый с новой строки (в строке наборы чисел через пробел). Необходимо выбрать и отобразить на экране уникальные числа, присутствующие и в первом и во втором списках одновременно. Результат выведите на экран в виде строки чисел, записанных по возрастанию через пробел, используя команду (здесь s - это множество):

```
print(*sorted(s))
```

P. S. О функции sorted мы еще будем говорить, а также об операторе *. Пока просто запомните такую возможность сортировки и вывода произвольных коллекций на экран.

Входные данные:

8 11 12 15 -2

4 11 10 15 -5 1 -2

Выходные данные:

-2 11 15

```
s1 = set(map(int, input().split()))

s2 = set(map(int, input().split()))


s = s1 & s2

print (*sorted(s))
```

Задание 2. 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 списке, но отсутствующие во втором. Результат выведите на экран в виде строки чисел, записанных по возрастанию через пробел.

Входные данные:

8 5 3 5 -3 1

1 2 3 4

Выходные данные:

-3 5 8

```
lst = set([int(x) for x in input(). split()])
lst_2 = set([int(x) for x in input(). split()])
love = lst - lst_2
```

Задание 3. 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.

Входные данные:

1 2 3 4 5

4 5 6 7 8

Выходные данные:

1 2 3 6 7 8

```
l = set(map(int, input().split()))
```

```
m = set(map(int, input().split()))
k = l|m
n = (l&m)
s = k-n
print(*sorted
```

Задание 4. Вводятся два списка городов каждый с новой строки (в строке названия через пробел). Необходимо сравнить их между собой на равенство по уникальным (не повторяющимся) городам. Если списки содержат одни и те же уникальные города, то вывести на экран ДА, иначе - НЕТ.

Входные данные:

Москва Тверь Уфа Казань Уфа Москва

Уфа Тверь Москва Казань

Выходные данные:

ДА

```
set_1 = set(input().split()) # Получаем пользовательские данные
set_2 = set(input().split()) # Получаем пользовательские данные
print("ДА" if set_1 == set_2 else "НЕТ")
```

Задача 5. На вход программе подается строка, состоящая из цифр. Необходимо определить, верно ли, что в ее записи ни одна из цифр не повторяется?

```
s = input()

print(('NO','YES')[len(s) == len(set(s))])
```

Задача 6. На вход программе подаются две строки, состоящие из цифр. Необходимо определить, верно ли, что в записи этих двух строк используются все десять цифр?

```
print(('NO', 'YES')[len(set(input() + input())) == 10])
```

Задача 7. На вход программе подается строка, состоящая из трех слов. Верно ли, что для записи всех трех слов был использован один и тот же набор букв?

```
a, b, c = input().split()

print(['NO', "YES"][set(a) == set(b) == set(c)])
```

Задача 8. Необходимо написать программу для проверки пароля на безопасность, в данном случае необходимо соблюсти хотя бы три критерия:

Длина пароля не менее 5 символов

Содержит буквы латинского алфавита как в верхнем, так и в нижнем регистре

Хотя бы одну цифру от 0 до 9

Хотя бы один спец.символ: "@,#,%,&

```
password = '1aA#'

conditions = (
    len(password) >= 5,
    any(map(str.islower, password)) and any(map(str.isupper, password)),
    set(password) & set("1234567890"),
    set(password) & set("@#%&"),
)

if sum(map(bool, conditions)) >= 3:
    print("Пароль достаточно сложный")
else:
    print('Пароль недостаточно сложный')
```

Генераторы множеств и генераторы словарей

На этом занятии мы с вами поговорим о генераторах множеств и словарей. Ранее, я вам уже рассказывал о генераторах списков, когда мы их создавали, используя синтаксис:

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

Например:

```
a = [x ** 2 for x in range(1, 5)]
print(a)
```

Далее, я буду полагать, что вы помните и знаете этот материал. Так вот, те же самые конструкции можно определять и для множеств со словарями. Для этого достаточно вместо квадратных скобок прописать фигурные:

```
a = {x ** 2 for x in range(1, 5)}
```

и вместо списка получим уже множество. Почему было сформировано именно множество, а не словарь? Как мы помним, множество представляет собой набор отдельных значений, а в словаре дополнительно еще прописываются ключи. Здесь же, при генерации мы получаем серию значений, поэтому, такая коллекция в Python воспринимается именно как множество.

А вот если мы будем генерировать последовательность с ключами, например, так:

```
a = {x: x ** 2 for x in range(1, 5)}
```

то получим уже словарь.

Где такие генераторы могут использоваться? Давайте представим, что нам нужно выделить уникальные значения из списка:

```
d = [1, 2, '1', '2', -4, 3, 4]
```

Причем, все элементы следует привести к целым числам перед записью в множество. Для решения этой задачи удобно воспользоваться генератором множества:

```
a = {int(x) for x in d}
```

Видите, как легко и просто, в одну строчку мы это сделали. Мало того, генераторы работают быстрее циклов. Поэтому реализация задачи в виде:

```
set_d = set()
for x in d:
    set_d.add(int(x))
print(set_d)
```

будет и более громоздкой и более медленной. Поэтому там, где это возможно, лучше использовать соответствующие генераторы.

То же самое и со словарем. Допустим, нам нужно все его ключи записать заглавными буквами, а значения представить целыми числами:

```
m = {"неудовл.": 2, "удовл.": 3, "хорошо": '4', "отлично": '5'}
```

Опять же используем генератор:

```
a = {key.upper(): int(value) for key, value in m.items()}
```

и на выходе получаем искомый словарь. Видите, как элегантно это можно сделать с использованием генераторов.

Также, как и со списками, в генераторах множеств и словарей можно использовать условия. Например, мы хотим в множество добавить только положительные числа из списка:

```
d = [1, 2, '1', '2', -4, 3, 4]
```

Также предварительно их преобразовываем в числа и делаем проверку:

```
a = {int(x) for x in d if int(x) > 0}
```

Или, со словарем. Поменяем в нем местами ключи и значения и поместим в него только отмети от 2 до 5:

```
m = {"безнадежно": 0, "убого": 1, "неудовл.": 2, "удовл.": 3, "хорошо": '4', "отлично": '5'}
a = {int(value): key for key, value in m.items() if int(value) >= 2 and int(value) <= 5}
```

Вот так просто и легко можно использовать генераторы множеств и словарей в своих программах. Основное их преимущество перед обычными циклами – более высокая скорость работы и наглядность текста программы. Поэтому, если циклы можно относительно легко заменить генераторами, то стоит это делать. Также в генераторах множеств и словарей можно применять и вложенные схемы, когда один генератор вложен в другой. Причем, можно комбинировать генераторы списков с генераторами множеств и словарей. Делается это все по аналогии с вложенными генераторами списков, о которых мы с вами уже говорили.

Задания на генераторы множеств и словарей

Задание 1. Вводится строка со списком оценок, например:

2 неудовлетворительно удовлетворительно хорошо отлично

Первая цифра - это числовое значение первой оценки. Остальные оценки имеют возрастающие на 1 числа. С помощью генератора словарей необходимо сформировать словарь d, где ключами будут выступать числа, а значениями - слова.
Например:

d = {2: 'неудовлетворительно', 3: 'удовлетворительно', 4: 'хорошо', 5: 'отлично'}

Вывести на экран значение сформированного словаря с ключом 4.

Входные данные:

1 ужасно неудовлетворительно удовлетворительно прилично отлично

Выходные данные:

прилично

Задание 2. На автомойку в течение квартала заезжали машины. Их гос. номера фиксировались в журнале, следующим образом (пример):

E220CK
A120MB

B101AA
E220CK
A120MB

На основе такого списка через генератор множеств сформировать список уникальных машин. На экран вывести число уникальных машин.

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Входные данные:

А323ГД

Д456ВВ

Б001ББ

Д456ВВ

С111СС

Выходные данные:

4

```
lst = ['E220CK','A120MB','B101AA','E220CK','A120MB']

lst_car = set(x for x in lst)
print (len(lst_car))
```

или

```
s = {x for x in lst_in} print(len(s))
```

Задание 3. Вводится текст в одну строчку со словами через пробел. С помощью генератора множеств сформировать множество из уникальных слов без учета регистра и длина которых не менее трех символов. Вывести на экран размер этого множества.

Входные данные:

Хижина изба машина и снова хижина машина

Выходные данные:

4

```
s = 'Хижина изба машина и снова хижина машина'
s = s.lower().split()
print(s)
set_out = {x for x in s if len(x)>=3}
print (len(set_out))
```

```
nt(len({i for i in input().lower().split() if len(i) > 2}))
```

Задание 4. Вводится текст в одну строчку со словами через пробел. Используя генераторы множеств и словарей, сформировать словарь в формате:

{слово_1: количество_1, слово_2: количество_2, ..., слово_N: количество_N}

То есть, ключами выступают уникальные слова (без учета регистра), а значениями - число их встречаемости в тексте. На экран вывести значение словаря для слова (союза) 'и'. Если такого ключа нет, то вывести 0.

Входные данные:

И что сказать и что сказать и нечего и точка

Выходные данные:

4

Задание 4. Вводится список книг книжного магазина в формате:

<автор 1>:<название 1>
...
<автор N>:<название N>

Авторы с названиями могут повторяться. Необходимо, используя генераторы, сформировать словарь с именем d вида:

{'автор 1': {'название 1', 'название 2', ..., 'название M'}, ..., 'автор K': {'название 1', 'название 2', ..., 'название S'}}

То есть, ключами выступают уникальные авторы, а значениями - множества с уникальными названиями книг соответствующего автора.

На экран ничего выводить не нужно, только сформировать словарь обязательно с именем d - он, далее будет проверяться в тестах!

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Входные данные:

Пушкин: Сказака о рыбаке и рыбке

Есенин: Письмо к женщине

Тургенев: Муму

Пушкин: Евгений Онегин

Есенин: Русь

Выходные данные:

True

Функция zip. Примеры использования

На прошлых занятиях мы с вами уже познакомились с функцией zip. На этом продолжим эту тему и поговорим о третьей часто применяемой функции:

zip(iter1 [, iter2 [, iter3] ...])

Она для указанных итерируемых объектов перебирает соответствующие элементы и продолжает работу до тех пор, пока не дойдет до конца самой короткой коллекции. Таким образом, она гарантирует, что на выходе будут формироваться наборы значений из всех переданных объектов.

Давайте в качестве простого примера возьмем два списка со значениями:

```
a = [1, 2, 3, 4]
b = [5, 6, 7, 8, 9, 10]
```

и передадим их в качестве аргументов функции zip():

```
z = zip(a, b)
print(z)
```

После запуска программы увидим, что переменная z ссылается на объект zip. Как вы уже догадались, это итератор, элементы которого перебираются функцией next():

```
print(next(z))
print(next(z))
```

Давайте переберем все пары через цикл for:

```
for x in z:
    print(x)
```

Здесь x является кортежем из соответствующих значений списков a и b. Причем последняя пара заканчивается числами (4, 8), когда функция zip() дошла до конца самого короткого списка.

Также всегда следует помнить, что функция zip() возвращает итератор. Это значит, что перебрать элементы можно только один раз. Например, повторный вызов оператора for для этой же функции:

```
for x in z:
    print(x)
```

ничего не возвратит, так как мы уже один раз прошлись по его элементам в предыдущем цикле.

Если нам все же нужно несколько раз обходить выделенные пары элементов, то их следует вначале преобразовать, например, в кортеж:

```
z = tuple(zip(a, b))
```

А, затем, обходить несчетное число раз. Но в этом случае увеличивается расход памяти, так как мы сохраняем все элементы в кортеже, а не генерируем «на лету» с помощью итератора. Так что, без особой необходимости делать преобразование к кортежу или списку не стоит.

В качестве перебираемых коллекций могут быть любые итерируемые объекты, например, строки. Если мы добавим строку:

```
c = "python"
```

и вызовем функцию zip() и с ней:

```
z = zip(a, b, c)
```

то в консоли увидим кортежи уже из трех значений, причем последнее будет соответствующим символом строки. Также, напомню, что оператор for можно записать с тремя переменными, если перебираемые значения являются кортежами:

```
for v1, v2, v3 in z:
    print(v1, v2, v3)
```

Иногда удобно кортеж сразу распаковать в отдельные переменные и использовать внутри цикла for.

Также все кортежи из функции zip() мы можем получить с помощью распаковки, используя запись:

```
z1, z2, z3, z4 = zip(a, b, c)
print(z1, z2, z3, z4, sep="\n")
```

Здесь неявно вызывается функция next() для извлечения всех элементов. Или, можно записать так:

```
z1, *z2 = zip(a, b, c)
print(z1, z2, sep="\n")
```

Тогда первое значение будет помещено в переменную z1, а все остальные – в список z2.

И в заключение этого занятия я покажу вам еще одно интересное преобразование, которое можно выполнить с помощью функции zip(). Если преобразовать итератор в список:

```
z = zip(a, b, c)
lz = list(z)
```

а, затем, распаковать его элементы с помощью оператора *:

```
print(*lz)
```

то мы получим уже не список, а четыре кортежа, следующих друг за другом. Если передать их в таком виде на вход функции zip():

```
t1, t2, t3 = zip(*lz)
print(t1, t2, t3, sep="\n")
```

то на выходе будут сформированы три кортежа из соответствующих элементов этих четырех. Вообще, это эквивалентно записи:

```
t1, t2, t3 = zip((1, 5, 'p'), (2, 6, 'y'), (3, 7, 't'), (4, 8, 'h'))
```

Но те же самые кортежи мы можем получить гораздо проще, если распаковать непосредственно итератор z:

```
t1, t2, t3 = zip(*z)
```

Как видите, к итератору вполне применим оператор распаковки элементов и на практике иногда этим пользуются.

Надеюсь, из этого занятия вы узнали, как работает функция zip() и ее назначение. Для закрепления материала, как всегда, пройдите практические задания.

Задания на zip

Вводятся два списка целых чисел. Необходимо попарно перебрать их элементы и перемножить между собой. При реализации программы используйте функции zip и map. Выведите на экран первые три значения, используя функцию next. Значения выводятся в строчку через пробел. (Полагается, что три выходных значения всегда будут присутствовать).

Sample Input:
-7 8 11 -1 3

1 2 3 4 5 6 7 8 9 10

Sample Output:

-7 16 33

```
s1 = list(map(int,input().split()))

s2 = list(map(int,input().split()))


s4 = map(lambda x: x[0]*x[1] , zip(s1,s2))

for _ in range(3):

    print (next(s4), end = ' ')
```

```
s1 = '-7 8 11 -1 3'
s2 = '1 2 3 4 5 6 7 8 9 10'

s1 = [int(i) for i in s1.split()]
s2 = [int(i) for i in s2.split()]

print(s1)
print(s2)
```

```
s3= [i*j for i,j in zip(s1,s2)]

print(s3)
```

Вводится неравномерная таблица целых чисел. С помощью функции zip выровнять эту таблицу, приведя ее к прямоугольному виду, отбросив выходящие элементы. Вывести результат на экран в виде такой же таблицы чисел.

P. S. Для считывания списка целиком в программе уже записаны начальные строчки.

Input:

1 2 3 4 5 6

3 4 5 6

7 8 9

9 7 5 3 2

Output:

1 2 3

3 4 5

7 8 9
9 7 5

```
for i in zip(*zip(*lst_in)):
    print (*i, sep='')
```