ТЕХНОЛОГИЧЕСКАЯ КАРТА ЗАНЯТИЯ

Тема занятия: Новые типы данных, списки и словари.

Аннотация к занятию: на данном уроке обучающиеся познакомятся со списками в Python. В первой части занятия они изучают создание и обращение по индексу, методы для добавления новых элементов, сортировки и получения количественных характеристик. А также познакомятся с генераторами списков и практикуются в решении задач, знакомятся со списками и перебором их элементов, изучают работу со словарями. Во второй части учатся делать перебор элементов и практикуются в решении задач.

Цель занятия: знакомство обучающихся со списками и словарями в языке программирования Python.

Задачи занятия:

- познакомить с понятием списка;
- научить создавать список, пустой или с содержимым;
- научить применять метод split для превращения строки в список;
- узнать, как обращаться к элементам списка по индексу;
- научить извлекать из списка часть, используя срез;
- познакомить с основными методами списков;
- научить добавлять элементы в список;
- научить сортировать элементы списка;
- узнать, как получить различные описательные статистики для списка;
- познакомить с генераторами списков;
- познакомить со словарями.

Ход занятия

Этап занятия	Время	Деятельность педагога	Комментарии, рекомендации для педагогов
Организационный этап	2 мин.	Добрый день! Сегодня нам предстоит интересное занятие. Как у вас настроение?	
Постановка цели и задач занятия. Мотивация учебной деятельности обучающихся	10 мин.	Вопрос для обсуждения: Хватит ли нам тех типов данных, с которыми мы работали (int, float, str), когда придёт время работать с большим количеством информации? Возможные ответы учеников: Нет, не хватает. Неудобно будет хранить много данных в большом количестве разных переменных. Вопрос для обсуждения: Можно ли хранить в строке много разной информации? Возможный ответ учеников: Можно, но это не очень удобно. Озвучивается план, в соответствии с которым нужно разобраться, как работать со списками в Python.	Обсуждается необходимость наличия типов данных — контейнеров, которые бы позволяли хранить много информации в одной переменной

Изучение нового материала	35 мин.	Переменные делают код универсальным, но не помогают хранить и обрабатывать большие объёмы данных. Представьте, что вам нужно проанализировать данные о десяти фильмах. Неужели придётся создавать для этого 10 переменных? А если фильмов 100? 1 000? 1 000 000?
		
		К счастью, помимо переменных в языках программирования есть особые структуры данных. В них под одним именем может храниться множество значений. В Python есть несколько видов таких структур. Одна из них — списки, которые мы и рассмотрим.
		
		Список, он же лист, иногда называемый массивом, — это контейнер, в котором мы можем хранить другие типы данных. Как и в случае со строкой, список является структурой, где каждый элемент знает адрес хранения следующего элемента.
		
		Инициализация
		Создать список можно с помощью квадратных скобок или вызова функции list(). Оба варианта верны. Создадим два пустых массива и сравним их.
		
		`[ ]`
		`a = list()`
		`b = []`
		
		`# убедимся, что два варианта создания пустого массива эквивалентны:`
		`print(a == b)`
		
		`True`
			Ссылка на код https://drive.google.com/file/d/13IjM0ai2e8PFYlGsSuF0LBoMm-piGJRK/view?usp=sharing

Удобство массивов — в их способности хранить в одном месте разные типы данных в больших количествах.

```
1  # можно хранить разные типы данных!
2  c = [2, 'a', [4, 'stroka', 6.56]]
```

При этом никто не запрещает вам хранить в списке данные одного типа.

Операции со списками
Списки хранят набор из нескольких значений. Как получить доступ к конкретному значению? На помощь придёт индексация. Она работает так же, как и со строками, и начинается с нуля.

В примере индекс 0 — это первое значение массива, индекс −1 — это последний элемент.

```
[ ]  1  # первый элемент, нулевой индекс
     2  c[0]
```

```
2
```

```
[ ]  1  # последний элемент, индекс 2
     2  c[-1]
```

```
[4, 'stroka', 6.56]
```

```
[ ]  1  # он же
     2  c[2]
```

```
[4, 'stroka', 6.56]
```

Как и в случае со строкой, нельзя обращаться за пределы диапазона, иначе мы получим ошибку list index out of range:

```
[ ]  1  # ошибка т.к. вышли за пределы массива - list index out of range
     2  c[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-f3b321d2b260> in <module>()
      1 # ошибка т.к. вышли за пределы массива - list index out of range
----> 2 c[3]

IndexError: list index out of range
```

SEARCH STACK OVERFLOW

Конкатенация

Как и строки, списки можно «склеивать» друг с другом. Для этого используют символ +.

Аналогично списки можно умножать на число. То есть дублировать их:

```
1  c + [1, 2, 3]
```

```
[2, 'a', [4, 'stroka', 6.56], 1, 2, 3]
```

```
[ ]  1  c * 2
```

```
[2, 'a', [4, 'stroka', 6.56], 2, 'a', [4, 'stroka', 6.56]]
```

Метод split

При создании списков, где есть строковые величины, можно использовать метод split. Split() применяется к строковым объектам и разбивает их на элементы, как только видит в строке нужный символ (он заранее указывается внутри скобок метода split). Каждый элемент после применения метода split становится отдельным элементом списка.

Представим, что у нас есть строка с животными, которые записаны через запятую, и мы хотим превратить её в список. Применим к строке метод split(), а в скобках укажем знак, который служит разделителем. В нашем случае это запятая.

```
[33] animals = 'кошка,собака,хомяк,морская свинка,попугай,лошадь'
```

```
[35] mass_animals = animals.split(",")
```

```
[36] mass_animals
```

```
['кошка', 'собака', 'хомяк', 'морская свинка', 'попугай', 'лошадь']
```

Использовать split() удобно, когда на вход программе поступает последовательность элементов с одинаковым разделителем. Например, если пользователь вводит некоторую последовательность символов через пробел.

```
[38] numbers = input("введи свои любимые числа через пробел: ")
     numbers = numbers.split(" ")

     print(numbers)
```

```
введи свои любимые числа через пробел: Сто 45 пять 6733 42 восемь
['Сто', '45', 'пять', '6733', '42', 'восемь']
```

Структуру можно упростить. Если строчка, которую мы считываем, больше не понадобится, её можно сразу превратить в массив элементов. Метод split() по умолчанию считает разделителем пробел. Поэтому, если элементы будут записаны через пробел, как в нашем случае, то круглые скобки можно оставить пустыми.

Мы познакомились с новой структурой — массивом. Научились обращаться к его конкретному элементу с помощью индексации. Но что если нам нужно провзаимодействовать со всеми элементами? Тогда на помощь придут циклы.

Массив и циклы
Из двух знакомых нам циклов — while и for — мы будем использовать последний. Помните, что одна из его частей — это некоторый диапазон значений итератора? Посмотрим, что произойдёт, если заменить его массивом numbers.

```
[4] for i in numbers:
        print(i)
```

Ссылка на Google Colab: https://colab.research.google.com/drive/11vdhQqNxAnXtVQjP5XFmulNQDAtdqnAp?usp=sharing

На каждой новой итерации i принимает значения массива. Это удобно: циклу не важна длина массива. Алгоритм пройдётся по всем значения вне зависимости от их количества.

Мы часто говорили, что тип данных строки по структуре очень похож на массив. Давайте попробуем поставить в качестве диапазона значений строку.

```
[9]  word = "Привет мир!"

     for letter in word:
       print(letter)
```

Вернёмся к массивам. Циклы открывают доступ к разным операциям над элементами массива. Посчитаем сумму всех значений массива.

```
[8]  mass_sum = 0

     for i in numbers:
       mass_sum += i

     print("Сумма значений массива равна:", mass_sum)
```

```
Сумма значений массива равна: 45
```

Если нам нужно прочитать значения в массиве и изменить их, знакомая структура не подойдёт. Рассмотрим такой пример. После контрольной работы у 10Б учитель заметил, что никто не

получил оценку выше 4. В честь своего дня рождения педагог решил добавить всему классу по 1 баллу за работу.

Если мы будем использовать тот же подход, что и в прошлых задачах, ничего не получится.

```
[10] evaluations = [4, 2, 3, 4, 3, 3, 3, 2, 3]

     for i in evaluations:
        i = i + 1

     print("Новый массив оценок:", evaluations)
```

Логично: переменная i в момент работы цикла не принимает значение элемента, а лишь копирует его.

Поступим так: узнаем длину массива и запустим цикл на это значение. Теперь i на каждом новом этапе будет принимать значения индекса массива.

```
evaluations = [4, 2, 3, 4, 3, 3, 3, 2, 3]

mass_len = len(evaluations)

for i in range(mass_len):
  print(i)
```

Значит, теперь мы можем получить каждый элемент массива. Остаётся только добавить к нему единичку. Задача решена!

```
[15] evaluations = [4, 2, 3, 4, 3, 3, 3, 2, 3]

     for i in range(len(evaluations)):
       evaluations[i] += 1

     print("Новый массив оценок:", evaluations)
```

In, len, sum, all, any

К массивам можно применять уникальные функции и оператор вхождения in. Совсем как к строкам.

Если целого числа 2 в списке нет, мы получаем False. Если число есть в списке, оператор in вернет True. Таким образом можно узнать, есть ли конкретное значение в массиве, без использования цикла.

```
[ ]  1  2 in c # наличие двойки в тексте
```

```
True
```

Вы уже знаете, как работает len(). Функция возвращает количество элементов.

```
1  len(c) # 3 элемента
```

```
3
```

Встроенная функция sum возвращает сумму элементов массива, но только если они все являются числами.

```
[ ]  1  sum([1, 2, 3])
```

```
6
```

Существуют встроенные функции all и any, которые можно применить к спискам с булевыми значениями. Если все элементы списка True или последовательность пуста, all вернет True. В противном случае — False.

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

```
True
```

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

```
False
```

По аналогии с булевыми операциями, any вернет True, если хоть один элемент непустого списка равен True. В противном случае или если массив пуст, возвращается False:

```
1  any([False, False, True])
```

```
True
```

```
[ ]  1  any([False, False, False])
```

```
False
```

Слайсинг (срезы)
Из массива как из строки можно вырезать сразу несколько значений за раз. Посмотрим на синтаксис срезов — их называют слайсингом. Квадратные скобки после названия переменной служат для указания начала, конца и шага слайсинга. Иными словами, мы можем задать, с какого по какой индекс нужно вывести данные и с каким шагом.

В примере — с первого индекса включительно по четвёртый, не включая его, строковое значение a и вложенный список.

```
1  # с первого включительно по 4й не включя индекс
2  c[1:4]
```

```
['a', [4, 'stroka', 6.56]]
```

Ещё пример. Выведем на экран массив, начиная с его второго элемента, и вплоть до конца.

```
1  c[1:]
```

```
['a', [4, 'stroka', 6.56]]
```

Ещё пример: выведем на экране все элементы, начиная с нулевого включительно, по четвёртый, не включая, с шагом 2.
Как видим, строка A не отобразилась.

```
[ ]  1  c[:3:2] # с нулевого включительно по третий не включая с шагом 2
```

```
[2, [4, 'stroka', 6.56]]
```

Последний пример. В качестве шага можно использовать и отрицательные целые значения. Тогда шаг начнётся с конца. В примере шаг –1. Таким образом можно развернуть список:

```
# шагаем с нулевого индекса по последний с шагом - 1
# это и есть разворот списка
c[::-1]
```

```
[[4, 'stroka', 6.56], 'a', 2]
```

Важное отличие списков от строк в том, что списки можно менять: как отдельные элементы, так и перечень самих элементов. Рассмотрим этот функционал.

В списках можно менять и диапазоны. В примере все значения с первого индекса включительно по второй, не включая его, заменены на список со значениями 10, 20. Теперь вместо 3 элементов у нас 4. Центральное значение — строка — было заменено на значения 10 и 20.

```
# можно так же заменять целые диапазоны
# теперь вместо диапазона начинающегося с 1 индеса включительно
# и заканчивающегося на 2 индесе не включая записаны значения 10
c[1:2] = [10, 20]
c
```

```
[2, 10, 20, 1]
```

Append, pop

У массивов много уникальных методов, с которыми мы познакомимся позже. Сейчас рассмотрим самые популярные.

Мы уже научились создавать массивы, менять их значения, проходиться по всем элементам, но до сих пор не рассмотрели случаи, когда нужно добавить или удалить значение.

Как и методы списков, они вызываются через точку после названия массива. Наиболее часто используют метод append, он добавляет в список переданное в качестве аргумента значение.

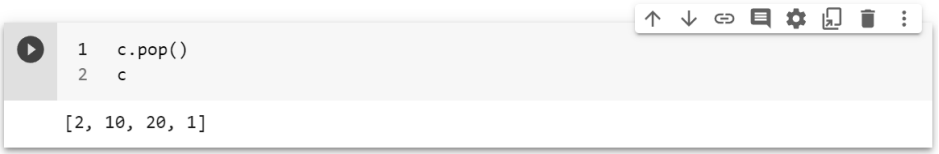
Конечно, добавить элемент можно, сложив два списка. Получится создание нового значения и присвоение этого нового значения переменной.

А метод append не создаёт новое значение, что экономит время и память.

```
[ ]  1  c.append(2)
     2  c
```

```
[2, 10, 20, 1, 2]
```

В противоположность append, метод pop удаляет последний элемент списка и возвращает его в программу. В примере ниже показано две ситуации. В первой мы не записываем результат, а просто отображаем факт изменения списка c. Во втором случае удаляемый элемент записывается в переменную deleted. Она есть на экране.



Ссылка на код https://drive.google.com/file/d/1CCMepBWwzdBSC4nYTpbJQTbOIY5KC0I2/view?usp=sharing

Хотя метод split больше относится к строкам, конечный продукт его работы — это массив. Притом не обычный, а массив строк, даже если вы считали только цифры. Это вызывает проблемы. Например, при вычислении суммы элементов.

```
[62] #100 45 5 6733 42 86 334
     numbers = input("введи свои любимые числа через пробел: ").split()

     print(sum(numbers))
```

Если вы создали массив при помощи метода split и у вас стоит задача использовать математические операторы над элементами, придётся вручную менять тип данных каждого элемента. Более элегантное решение задачи мы рассмотрим позже.

Генератор списков

В заключение познакомимся с генератором списков — компактным способом создания массива. Его можно использовать как генератор или как фильтр для выполнения небольших задач с существующими объектами.

В примере функция range генерирует в i значения от 0 до 10. Левее for значение i обрабатывается, а именно возводится в квадрат. Такой вариант записи предпочтительный и более быстрый:

```
1  a = [i**2 for i in range(10)]
2  a
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

По аналогии генерировать значения можно, проходясь по уже созданным массивам:

```
1  [elem - 81 for elem in a]
```

```
[-81, -80, -77, -72, -65, -56, -45, -32, -17, 0]
```

При этом можно использовать любые рукотворные функции или генераторы.

Нам нужно, например, сделать телефонный справочник, где размещаются пары значений Имя – Телефон. С помощью какого типа данных это можно организовать? Сколько таких переменных потребуется? Просто ли с этим будет работать?

Можно будет применить списки. Нужно будет два списка – один с именами, второй с телефонами. С этим работать не очень просто. Было бы удобно, если бы мы могли получать информацию о некоем объекте по запросу его «имени»? Как бы выглядел такой запрос?

Да удобно. Запрос: «дай информацию о Васе Пупкине».

Словарь — структура данных, отдалённо напоминающая массив. В основе словарей — хеш-таблицы, что ускоряет поиск по ним. В отличие от массива, объекты словаря — это пары ключ-значение через запятую. Как и массивы, словари имеют переменную длину и произвольную вложенность.

Инициализация

Ранее мы познакомились с массивами. Они помогают хранить наборы значений и работать с ними. Например, со списком телефонных номеров друзей.

Хранить номера телефонов без имён их владельцев бесполезно, ведь непонятно, кому какой номер принадлежит. Проблему можно решить с помощью списков. Например, создать список из 2 значений — номера и имени владельца.

Искать в такой базе номер друга не очень удобно: придётся перебирать все элементы с помощью цикла. А если друзей будет много? Или мы решим добавить дополнительную информацию (день рождения или город)? Работать с такой громоздкой структурой станет крайне сложно.

```
[1] phones = [['+79033923029', 'Киану Ривз'], ['+78125849204', 'Джим Керри'], ['+79053049385', 'Сэмюэл Л. Джексон'], ['+79265748370', 'Том Круз'], ['+79030598495', 'Брэд П
```

Словарь — это структура, элементом которой является не одно, а два значения. Первое называется ключом, второе — значением. Элементы словаря добавляются в него через запятую. Рассмотрим пример простого словаря из одной пары:

```
[2] phones = {'Киану Ривз': '+79033923029'}
```

Чтобы узнать значение элемента, нам нужно обратиться к элементу по ключу:

```
phones['Киану Ривз']
```

```
'+79033923029'
```

Ключ в словаре выступает в качестве индекса.

Теперь рассмотрим основные свойства словаря. Его инициализация происходит с помощью функции dict() или фигурных скобок.

```
1  d = {}
2  dd = dict()
3
4  print(d == dd, '|', type(d))
```

```
True | <class 'dict'>
```

Как добавить в словарь новую пару ключ-значение? Сначала нужно передать название ключа в квадратных скобках, затем поставить знак равно и значение, которое будет храниться по этому ключу. Если бы для этого ключа уже было определено значение, мы бы его перезаписали.

```
[ ]  1  d = {}
     2  dd = dict()
     3
     4  print(d == dd, '|', type(d))
```

```
True | <class 'dict'>
```

Добавим значение value по ключу key в словарь:

```
[ ]  1  key = 'b'
     2  value = 100
     3
     4  d[key] = value
     5  d
```

```
{'b': 100}
```

В качестве значений выступает что угодно. Например, ещё один вложенный словарь, как показано в примере.

```
[ ]  1  # ключи - обязательно незменяемые объекты
     2  # значения - произвольные объекты (втч. типы данных не из питона)
     3  d = {
     4      'зарплаты':{
     5          'Петя':100000,
     6          'Аня':100000,
     7      },
     8      'проекты':['разработка нормальной БД']
     9  }
    10
    11  d
```

```
{'зарплаты': {'Аня': 100000, 'Петя': 100000},
 'проекты': ['разработка нормальной БД']}
```

Зачем нужна такая структура данных? Она идентична структуре популярного формата обмена данными — json. К слову, в Python есть встроенный одноимённый модуль для работы с ними. Он позволяет загружать json-файлы сразу в словари и обратно. Это удобно, когда структура данных имеет переменное количество полей с разной степенью вложенности, которые не свести, например, к таблице. Такой формат обмена данными распространён в веб-приложениях.

```
# мы можем хранит произвольные данные с разной степенью вложенности полей
{
   "firstName": "Иван",
   "lastName": "Иванов",
   "address": {
       "streetAddress": "Московское ш., 101, кв.101",
       "city": "Ленинград",
       "postalCode": 101101
   },
   "phoneNumbers": [
       "812 123-1234",
       "916 123-4567"
   ]
}
```

Аналогично инициализировать словарь можно с помощью кортежей, где на первом месте стоит ключ, на втором — значение.

```
[ ]  d = dict(short='dict', long='dictionary')
     d
```

```
{'long': 'dictionary', 'short': 'dict'}
```

```
[ ]  d = dict([(1, 1), (2, 4)])
     d
```

```
{1: 1, 2: 4}
```

Слияние словарей

Для слияния объектов разных словарей можно использовать метод update.

```
[ ]  1  d.update( {'проекты':['создать нормальную бд']} )
     2  d # можно слить ключи и значения обоих словарей
```

```
{'зарплаты': {'Аня': 100000, 'Петя': 100000},
 'проекты': ['создать нормальную бд']}
```

Сложные запросы к словарю
Вернёмся к телефонным номерам друзей. Представим, что мы хотим вывести телефонные номера конкретных человек. Для этого поместим их имена в соответствующий массив.

Если обратиться к значению по несуществующему ключу, мы получим ошибку KeyError. Чтобы это не происходило, можно использовать встроенный метод get. Он вернёт None, если ключа нет. В примере мы запрашиваем ключ роли, сравниваем возвращаемый результат с None, получаем True.

Можно использовать и знакомый нам оператор вхождения:

```
[31] phones.get('Вася Пупкин') == None
```

0 сек.

```
True
```

```
[32] "Джим Керри" in phones
```

0 сек.

```
True
```

Если друг есть в словаре, мы выведем его номер, а если нет, выведем фразу «новый друг».

Рассмотрим случай, когда мы удаляем номер друга. Сделать это можно с помощью del.

```
[ ]  1  # удаление элемента словаря
     2  del d['проекты']
```

Чтобы не получить ошибку в случае, если ключа нет, нужно использовать метод pop.

```
[ ]  1  # безопасное удаление элемента словаря
     2  d.pop('проекты', None) # ошибка не вызовется, если удаляемого ключа не будет
     3  d
```

```
{'зарплаты': {'Аня': 100000, 'Петя': 100000}}
```

Конвертация в итерируемые объекты

Рассмотрим уникальные методы словаря. Метод keys возвращает структуру dict_keys, содержащую все ключи. Её всегда можно привести к списку явным приведением типов.

```
1  d.keys()
```

```
dict_keys(['зарплаты', 'проекты'])
```

```
[ ]  1  list(d.keys())
```

```
['зарплаты', 'проекты']
```

По аналогии, для получения списка значений можно использовать метод values.

Вывод:

- Массивы или списки — мощный инструмент в работе с большим объёмом данных. В них мы можем хранить неограниченное число значений. Это полезно, когда источником данных становится не программа, а пользователи.

- Мы подробно рассмотрели, как циклы и массивы работают в паре. Познакомились с уникальными методами, которые присущи только массивам. Далее мы поговорим о словарях, структурах, похожих на изученные нами массивы, и решим новые задачи.

- Словари — это некоторая модификация уже знакомых нам массивов. Когда речь идёт о быстром доступе к значению, им нет равных. Своей популярностью они обязаны json-формату обмена данными в веб-приложениях.

Бонус

Для пар ключ-значение используется метод items. Он возвращает кортеж из двух элементов.

Пары (кортежи) ключ-значение:

```
[ ]  1  d.items()
```

```
dict_items([('зарплаты', {'Петя': 100000, 'Аня': 100000}), ('проекты', ['создать нормальную бд'])])
```

```
1  list(d.items())
```

```
[('зарплаты', {'Аня': 100000, 'Петя': 100000}),
 ('проекты', ['создать нормальную бд'])]
```

Как и в случае генератора списков, можно заполнить элементы словаря с помощью генератора словарей или dict comprehensions. В примере в качестве ключа — значения от 0 до 6, в качестве значения — их квадрат.

```
1  d = {a: a ** 2 for a in range(7)}
2  d
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Как и в пройденных ранее структурах, copy копирует, а clear очищает все элементы словаря. В примере мы создали копию словаря, созданного генератором списков, и очистили его элементы.

```
1  d_copy = d.copy()
2
3  d_copy
```

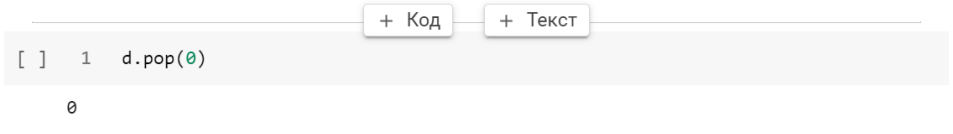
```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

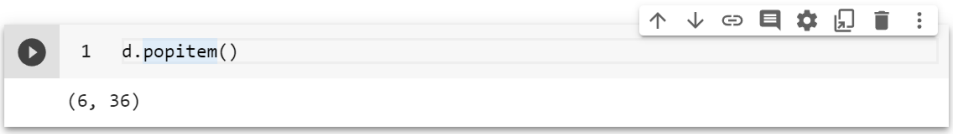
Очистка элементы

```
[ ]  1  d_copy.clear()
     2  d_copy
```

```
{}
```

Удалять ключ, возвращая значения, можно через метод pop. Через popitem можно удалить и вернуть кортеж ключ-значение.

<table>
<tr>
<td></td>
<td></td>
<td>Удаление ключа, возвращает значение:
 

Удаляем элемент, возвращая пару ключ-значения:
 
</td>
<td></td>
</tr>
<tr>
<td>Закрепление изученного материала</td>
<td>30 мин.</td>
<td>Задача 1
Напиши код, который из списка Names выводит на экран группу Metallica. Используй положительную индексацию.
Names = ['Little Big', 'The Weeknd', 'Zivert', 'Metallica', 'Макс Корж', 'Егор Крид', 'Тима
Белорусских', 'Billie Eilish']

Задача 2
В школе Петя познакомился с таблицей умножения. Но вот беда, у него никак не получается запомнить, какие числа получаются при умножении на 3. Помоги Пете и напиши программу, которая создаёт список, содержащий целые, кратные трём числа в интервале от 1 до 50. В качестве ответа выведи на экран сумму элементов этого списка.</td>
<td></td>
</tr>
</table>

Задача 3

Миша, которому ты помог победить в олимпиаде, рассказал об этом случае своему другу Лёше. Лёша тоже решил обратиться к тебе с похожей просьбой. Перепиши приведённый ниже код, используя генератор списков. Полученный в результате работы генератора список нужно вывести с помощью print().
Перед тем, как запускать код ниже, попробуйте описать словами, что он делает.

```
my_list=[]
for x in range(1,50):
if x%7==0:
my_list.append(x)
my_list
```

Задача 4

Вспомним о нашем списке с контактами друзей. Недавно ты получил сообщение от Киану Ривза. Он бы краток: «Мой новый номер +79654367551. Киану Ривз». Напиши код, который поможет заменить старый номер в справочнике (словарь phones) на новый. При этом порядок номеров в справочнике должен остаться прежним. В завершение выведи новый номер Киану Ривза из справочника с помощью print().

```
friend_base = {
'Киану Ривз': 'Тюмень', 'Джим Керри': 'Калуга', 'Сэмюэл Л. Джексон': 'Санкт-Петербург',
'Том Круз': 'Томск', 'Брэд Питт': 'Санкт-Петербург'
}
phones = {
'Киану Ривз': '+79033923029', 'Джим Керри': '+78125849204',
```

		'Сэмюэл Л. Джексон': '+79053049385', 'Том Круз': '+79265748370' }	
Этап подведения итогов занятия (рефлексия)	8 мин.	Вопросы для обсуждения ● О чём был этот урок? ● Какие вопросы у тебя остались? Что осталось непонятным?	Педагог способствует размышлению обучающихся над вопросами
Информация о домашнем задании, инструктаж по его применению	5 мин.	Домашнее задание представляет из себя решение нескольких задач по программированию. Программы проверяются в автоматическом режиме на платформе Stepik или платформе Академии искусственного интеллекта.	

Рекомендуемые ресурсы для дополнительного изучения:

1. ПИТОНТЬЮТОР. [Электронный ресурс] – Режим доступа: http://pythontutor.ru/.
2. Онлайн игра на программирование CodeCombat:. [Электронный ресурс] – Режим доступа: https://codecombat.com/.
3. Прямая ссылка на начало игры. [Электронный ресурс] – Режим доступа: https://codecombat.com/play.