Лабораторная работа №3. Сложные типы данных.

Строки в Python - упорядоченные последовательности символов, используемые для хранения и представления текстовой информации, поэтому с помощью строк можно работать со всем, что может быть представлено в текстовой форме.

Это первая часть о работе со строками, а именно о литералах строк.

Литералы строк

Работа со строками в Python очень удобна. Существует несколько литералов строк, которые мы сейчас и рассмотрим.

Строки в апострофах и в кавычках

```
S = 'spam"s'

S = "spam's"
```

Строки в апострофах и в кавычках - одно и то же. Причина наличия двух вариантов в том, чтобы позволить вставлять в литералы строк символы кавычек или апострофов, не используя экранирование.

Экранированные последовательности - служебные символы

Экранированные последовательности позволяют вставить символы, которые сложно ввести с помощью генератора случайных чисел.

Экранированная последовательность	Назначение
\n	Перевод строки
\a	Звонок
\b	Забой
\f	Перевод страницы
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\N{id}	Идентификатор ID базы данных Юникода
\uhhhh	16-битовый символ Юникода в 16-ричном представлении

Экранированная последовательность	Назначение
\Uhhhh…	32-битовый символ Юникода в 32-ричном представлении
\xhh	16-ричное значение символа
\ooo	8-ричное значение символа
\0	Символ Null (не является признаком конца строки)

"Сырые" строки - подавляют экранирование

Если перед открывающей кавычкой стоит символ 'r' (в любом регистре), то механизм экранирования отключается.

```
S = r'C:\newt.txt'
```

Но, несмотря на назначение, "сырая" строка не может заканчиваться символом обратного слэша. Пути решения:

```
S = r'\n\n\\'[:-1]

S = r'\n\n' + '\\'

S = '\\n\\n'
```

Строки в тройных апострофах или кавычках

Главное достоинство строк в тройных кавычках в том, что их можно использовать для записи многострочных блоков текста. Внутри такой строки возможно присутствие кавычек и апострофов, главное, чтобы не было трех кавычек подряд.

```
>>> c = '''это очень большая

... строка, многострочный

... блок текста'''

>>> c

'это очень большая\nстрока, многострочный\nблок текста'
```

```
>>> print(c)

это очень большая

строка, многострочный

блок текста
```

Базовые операции

- Конкатенация (сложение)

```
>>>
```

```
>>> S1 = 'spam'

>>> S2 = 'eggs'

>>> print(S1 + S2)

'spameggs'
```

- Дублирование строки

```
>>>
```

```
>>> print('spam' * 3)

spamspamspam
```

- Длина строки (функция len)

```
>>>
```

```
>>> len('spam')

4
```

- Доступ по индексу

```
>>>
```

```
>>> S = 'spam'

>>> S[0]

's'

>>> S[2]

'a'

>>> S[-2]

'a'
```

Как видно из примера, в Python возможен и доступ по отрицательному индексу, при этом отсчет идет от конца строки.

- Извлечение среза

Оператор извлечения среза: [X:Y]. X – начало среза, а Y – окончание;

символ с номером Y в срез не входит. По умолчанию первый индекс равен 0, а второй - длине строки.

```
>>>
```

```
>>> s = 'spameggs'

>>> s[3:5]

'me'

>>> s[2:-2]

'ameg'

>>> s[:6]

'spameg'
```

```
>>> s[1:]

'pameggs'

>>> s[:]

'spameggs'
```

Кроме того, можно задать шаг, с которым нужно извлекать срез.

```
>>>
```

```
>>> s[::-1]

'sggemaps'

>>> s[3:5:-1]

''

>>> s[2::2]

'aeg'
```

Другие функции и методы строк

При вызове методов необходимо помнить, что строки в Python относятся к категории неизменяемых последовательностей, то есть все функции и методы могут лишь создавать новую строку.

```
>>>
```

```
>>> s = 'spam'

>>> s[1] = 'b'

Traceback (most recent call last):

  File "", line 1, in

    s[1] = 'b'
```

```
TypeError: 'str' object does not support item assignment

>>> s = s[0] + 'b' + s[2:]

>>> s

'sbam'
```

Поэтому все строковые методы возвращают новую строку, которую потом следует присвоить переменной.

Таблица "Функции и методы строк"

Функция или метод	Назначение
S = 'str'; S = "str"; S = '''str'''; S = """str"""	Литералы строк
S = "s\np\ta\nbbb"	Экранированные последовательности
S = r"C:\temp\new"	Неформатированные строки (подавляют экранирование)
S = b"byte"	Строка байтов
S1 + S2	Конкатенация (сложение строк)
S1 * 3	Повторение строки
S[i]	Обращение по индексу
S[i:j:step]	Извлечение среза
len(S)	Длина строки
S.find(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или -1
S.rfind(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или -1
S.index(str, [start],[end])	Поиск подстроки в строке. Возвращает номер первого вхождения или вызывает ValueError
S.rindex(str, [start],[end])	Поиск подстроки в строке. Возвращает номер последнего вхождения или вызывает ValueError
S.replace(шаблон, замена)	Замена шаблона
S.split(символ)	Разбиение строки по разделителю

Функция или метод	Назначение
S.isdigit()	Состоит ли строка из цифр
S.isalpha()	Состоит ли строка из букв
S.isalnum()	Состоит ли строка из цифр или букв
S.islower()	Состоит ли строка из символов в нижнем регистре
S.isupper()	Состоит ли строка из символов в верхнем регистре
S.isspace()	Состоит ли строка из неотображаемых символов (пробел, символ перевода страницы ('\f'), "новая строка" ('\n'), "перевод каретки" ('\r'), "горизонтальная табуляция" ('\t') и "вертикальная табуляция" ('\v'))
S.istitle()	Начинаются ли слова в строке с заглавной буквы
S.upper()	Преобразование строки к верхнему регистру
S.lower()	Преобразование строки к нижнему регистру
S.startswith(str)	Начинается ли строка S с шаблона str
S.endswith(str)	Заканчивается ли строка S шаблоном str
S.join(список)	Сборка строки из списка с разделителем S
ord(символ)	Символ в его код ASCII
chr(число)	Код ASCII в символ
S.capitalize()	Переводит первый символ строки в верхний регистр, а все остальные в нижний
S.center(width, [fill])	Возвращает отцентрованную строку, по краям которой стоит символ fill (пробел по умолчанию)
S.count(str, [start],[end])	Возвращает количество непересекающихся вхождений подстроки в диапазоне [начало, конец] (0 и длина строки по умолчанию)
S.expandtabs([tabsize])	Возвращает копию строки, в которой все символы табуляции заменяются одним или несколькими пробелами, в зависимости от текущего столбца. Если TabSize не указан, размер табуляции полагается равным 8 пробелам
S.lstrip([chars])	Удаление пробельных символов в начале строки
S.rstrip([chars])	Удаление пробельных символов в конце строки
S.strip([chars])	Удаление пробельных символов в начале и в конце строки

Функция или метод	Назначение
S.partition(шаблон)	Возвращает кортеж, содержащий часть перед первым шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий саму строку, а затем две пустых строки
S.rpartition(sep)	Возвращает кортеж, содержащий часть перед последним шаблоном, сам шаблон, и часть после шаблона. Если шаблон не найден, возвращается кортеж, содержащий две пустых строки, а затем саму строку
S.swapcase()	Переводит символы нижнего регистра в верхний, а верхнего – в нижний
S.title()	Первую букву каждого слова переводит в верхний регистр, а все остальные в нижний
S.zfill(width)	Делает длину строки не меньшей width, по необходимости заполняя первые символы нулями
S.ljust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя последние символы символом fillchar
S.rjust(width, fillchar=" ")	Делает длину строки не меньшей width, по необходимости заполняя первые символы символом fillchar
S.format(*args, **kwargs)	Форматирование строки

Что такое списки?

Списки в Python - упорядоченные изменяемые коллекции объектов произвольных типов (почти как массив, но типы могут отличаться).

Чтобы использовать списки, их нужно создать. Создать список можно несколькими способами. Например, можно обработать любой итерируемый объект (например, строку) встроенной функцией **list**:

```
>>> list('список')

['с', 'п', 'и', 'с', 'о', 'к']
```

Список можно создать и при помощи литерала:

```
>>> s = []  # Пустой список
```

```
>>> l = ['s', 'p', ['isok'], 2]

>>> s

[]

>>> l

['s', 'p', ['isok'], 2]
```

Как видно из примера, список может содержать любое количество любых объектов (в том числе и вложенные списки), или не содержать ничего.

И еще один способ создать список - это **генераторы списков**. Генератор списков - способ построить новый список, применяя выражение к каждому элементу последовательности. Генераторы списков очень похожи на цикл for.

```
>>> c = [c * 3 for c in 'list']

>>> c

['lll', 'iii', 'sss', 'ttt']
```

Возможна и более сложная конструкция генератора списков:

```
>>> c = [c * 3 for c in 'list' if c != 'i']

>>> c

['lll', 'sss', 'ttt']

>>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d !=
'a']

>>> c

['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

Но в сложных случаях лучше пользоваться обычным циклом for для генерации списков.

Функции и методы списков

Создать создали, теперь нужно со списком что-то делать. Для списков доступны основные встроенные функции, а также методы списков.

Таблица "методы списков"

Метод	Что делает
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет на i-ый элемент значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
list.pop([i])	Удаляет i-ый элемент и возвращает его. Если индекс не указан, удаляется последний элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (при этом поиск ведется от start до end)
list.count(x)	Возвращает количество элементов со значением x
list.sort([key=функция])	Сортирует список на основе функции
list.reverse()	Разворачивает список
list.copy()	Поверхностная копия списка
list.clear()	Очищает список

Нужно отметить, что методы списков, в отличие от строковых методов, изменяют сам список, а потому результат выполнения не нужно записывать в эту переменную.

```
>>> l = [1, 2, 3, 5, 7]

>>> l.sort()

>>> l

[1, 2, 3, 5, 7]

>>> l = l.sort()
```

```
>>> print(l)

None
```

И, напоследок, примеры работы со списками:

```
>>> a = [66.25, 333, 333, 1, 1234.5]

>>> print(a.count(333), a.count(66.25), a.count('x'))

2 1 0

>>> a.insert(2, -1)

>>> a.append(333)

>>> a

[66.25, 333, -1, 333, 1, 1234.5, 333]

>>> a.index(333)

1

>>> a.remove(333)

>>> a

[66.25, -1, 333, 1, 1234.5, 333]

>>> a.reverse()

>>> a

[333, 1234.5, 1, 333, -1, 66.25]

>>> a.sort()

>>> a
```

```
[-1, 1, 66.25, 333, 333, 1234.5]
```

Модуль array. Массивы в python

Модуль array определяет массивы в python. Массивы очень похожи на списки, но с ограничением на тип данных и размер каждого элемента.

Размер и тип элемента в массиве определяется при его создании и может принимать следующие значения:

Код типа	Тип в C	Тип в python	Минимальный размер в байтах
'b'	signed char	int	1
'B'	unsigned char	int	1
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Класс **array.array**(TypeCode [, инициализатор]) - новый массив, элементы которого ограничены TypeCode, и инициализатор, который должен быть списком, объектом, поддерживающий интерфейс буфера, или итерируемый объект.

array.typecodes - строка, содержащая все возможные типы в массиве.

Массивы изменяемы. Массивы поддерживают все списковые методы (индексация, срезы, умножения, итерации), и другие методы.

Методы массивов (array) в python

array.typecode - TypeCode символ, использованный при создании массива.

array.itemsize - размер в байтах одного элемента в массиве.

array.append(x) - добавление элемента в конец массива.

array.buffer_info() - кортеж (ячейка памяти, длина). Полезно для низкоуровневых операций.

array.byteswap() - изменить порядок следования байтов в каждом элементе массива. Полезно при чтении данных из файла, написанного на машине с другим порядком байтов.

array.count(x) - возвращает количество вхождений x в массив.

array.extend(iter) - добавление элементов из объекта в массив.

array.frombytes(b) - делает массив array из массива байт. Количество байт должно быть кратно размеру одного элемента в массиве.

array.fromfile(F, N) - читает N элементов из файла и добавляет их в конец массива. Файл должен быть открыт на бинарное чтение. Если доступно меньше N элементов, генерируется исключение EOFError , но элементы, которые были доступны, добавляются в массив.

array.fromlist(список) - добавление элементов из списка.

array.index(x) - номер первого вхождения x в массив.

array.insert(n, x) - включить новый пункт со значением x в массиве перед номером n. Отрицательные значения рассматриваются относительно конца массива.

array.pop(i) - удаляет i-ый элемент из массива и возвращает его. По умолчанию удаляется последний элемент.

array.remove(x) - удалить первое вхождение x из массива.

array.reverse() - обратный порядок элементов в массиве.

array.tobytes() - преобразование к байтам.

array.tofile(f) - запись массива в открытый файл.

array.tolist() - преобразование массива в список.

Вот и всё, что можно было рассказать про массивы. Они используются редко, когда нужно достичь высокой скорости работы. В остальных случаях массивы можно заменить другими типами данных: списками, кортежами, строками.

Зачем нужны кортежи, если есть списки?

- Защита от дурака. То есть кортеж защищен от изменений, как намеренных (что плохо), так и случайных (что хорошо).
- Меньший размер. Дабы не быть голословным:

>>>

```
>>> a = (1, 2, 3, 4, 5, 6)

>>> b = [1, 2, 3, 4, 5, 6]

>>> a.__sizeof__()
```

```
36

>>> b.__sizeof__()

44
```

- Возможность использовать кортежи в качестве ключей словаря:

>>>

```
>>> d = {(1, 1, 1) : 1}

>>> d

{(1, 1, 1): 1}

>>> d = {[1, 1, 1] : 1}

Traceback (most recent call last):

  File "", line 1, in

    d = {[1, 1, 1] : 1}

TypeError: unhashable type: 'list'
```

Как работать с кортежами?

С преимуществами кортежей разобрались, теперь встает вопрос - а как с ними работать. Примерно так же, как и со списками.

Создаем пустой кортеж:

>>>

```
>>> a = tuple() # С помощью встроенной функции tuple()

>>> a

()

>>> a = () # С помощью литерала кортежа
```

```
>>> a

()

>>>
```

Создаем кортеж из одного элемента:

```
>>>
```

```
>>> a = ('s')

>>> a

's'
```

Стоп. Получилась строка. Но как же так? Мы же кортеж хотели! Как же нам кортеж получить?

```
>>>
```

```
>>> a = ('s', )

>>> a

('s',)
```

Ура! Заработало! Все дело - в запятой. Сами по себе скобки ничего не значат, точнее, значат то, что внутри них находится одна инструкция, которая может быть отделена пробелами, переносом строк и прочим мусором. Кстати, кортеж можно создать и так:

```
>>>
```

```
>>> a = 's',

>>> a

('s',)
```

Но все же не увлекайтесь, и ставьте скобки, тем более, что бывают случаи, когда скобки необходимы.

Ну и создать кортеж из итерируемого объекта можно с помощью все той же пресловутой функции tuple()

>>>

```
>>> a = tuple('hello, world!')

>>> a

('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')
```

Операции с кортежами

Все операции над списками, не изменяющие список (сложение, умножение на число, методы index() и count() и некоторые другие операции). Можно также по-разному менять элементы местами и так далее.

Например, гордость программистов на python - поменять местами значения двух переменных:

```
a, b = b, a
```

Что такое множество?

Множество в python - "контейнер", содержащий не повторяющиеся элементы в случайном порядке.

Создаём множества:

>>>

```
>>> a = set()

>>> a

set()

>>> a = set('hello')

>>> a

{'h', 'o', 'l', 'e'}

>>> a = {'a', 'b', 'c', 'd'}
```

```
>>> a

{'b', 'c', 'a', 'd'}

>>> a = {i ** 2 for i in range(10)} # генератор множеств

>>> a

{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> a = {}  # А так нельзя!

>>> type(a)

<class 'dict'>
```

Как видно из примера, множества имеет тот же литерал, что и словарь, но пустое множество с помощью литерала создать нельзя.

Множества удобно использовать для удаления повторяющихся элементов:

```
>>>
```

```
>>> words = ['hello', 'daddy', 'hello', 'mum']

>>> set(words)

{'hello', 'daddy', 'mum'}
```

С множествами можно выполнять множество операций: находить объединение, пересечение...

- len(s) - число элементов в множестве (размер множества).
- x in s - принадлежит ли x множеству s.
- **set.isdisjoint**(other) - истина, если set и other не имеют общих элементов.
- **set == other** - все элементы set принадлежат other, все элементы other принадлежат set.
- **set.issubset**(other) или **set <= other** - все элементы set принадлежат other.
- **set.issuperset**(other) или **set >= other** - аналогично.
- **set.union**(other, ...) или **set | other | ...** - объединение нескольких множеств.
- **set.intersection**(other, ...) или **set & other & ...** - пересечение.

- **set.difference**(other, ...) или **set - other - ...** - множество из всех элементов set, не принадлежащие ни одному из other.
- **set.symmetric_difference**(other); **set ^ other** - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
- **set.copy**() - копия множества.

И операции, непосредственно изменяющие множество:

- **set.update**(other, ...); set |= other | ... - объединение.
- **set.intersection_update**(other, ...); set &= other & ... - пересечение.
- **set.difference_update**(other, ...); set -= other | ... - вычитание.
- **set.symmetric_difference_update**(other); set ^= other - множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
- **set.add**(elem) - добавляет элемент в множество.
- **set.remove**(elem) - удаляет элемент из множества. KeyError, если такого элемента не существует.
- **set.discard**(elem) - удаляет элемент, если он находится в множестве.
- **set.pop**() - удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.
- **set.clear**() - очистка множества.

frozenset

Единственное отличие set от frozenset заключается в том, что set - изменяемый тип данных, а frozenset - нет. Примерно похожая ситуация с списками и кортежами.

```
>>>
```

```
>>> a = set('qwerty')

>>> b = frozenset('qwerty')

>>> a == b

True

>>> True

True

>>> type(a - b)
```

```
<class 'set'>

>>> type(a | b)

<class 'set'>

>>> a.add(1)

>>> b.add(1)

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AttributeError: 'frozenset' object has no attribute 'add'
```

Справка по использованию генератора случайных чисел.

import random
random.random() — возвращает псевдослучайное число от 0.0 до 1.0
random.random()
0.07500815468466127

random.seed
random.seed(<Параметр>) — настраивает генератор случайных чисел на новую последовательность. По умолчанию используется системное время.
random.seed(20)
random.random()
0.9056396761745207
random.random()
0.6862541570267026
random.seed(20)
random.random()
0.9056396761745207

random.uniform
random.uniform(<Начало>, <Конец>) — возвращает псевдослучайное вещественное число в диапазоне от <Начало> до <Конец>:
random.uniform(0, 20)
15.330185127252884
random.uniform(0, 20)
18.092324756265473

random.randint

random.randint(<Начало>, <Конец>) — возвращает псевдослучайное целое число в диапазоне от <Начало> до <Конец>:

```
random.randint(1,27)
9
random.randint(1,27)
22
```

Задание.

Задание 1. Дана строка, заканчивающаяся точкой.

a. Вывести длину строки.
b. Подсчитать, сколько слов в строке.
c. Найти длину самого короткого слова и самого длинного слова.
d. Преобразовать ее, удалив каждый символ * и повторив каждый символ, отличный от *.

Задание 2. Варианты заданий на перемещение элементов списка. Элементы списка вводить с помощью генератора случайный чисел.

1. Скорректировать список $A=(a_1, a_2, ..., a_n)$, переписав в начало списка группу, содержащую наибольшее число подряд идущих положительных элементов.
2. В списке $A=(a_1, a_2, ..., a_n)$ все элементы, равные нулю, поставить сразу после максимального элемента данного списка.
3. В списке $A=(a_1, a_2, ..., a_n)$ все отрицательные элементы отправить в «хвост» списка.
4. В списке $A=(a_1, a_2, ..., a_n)$ все положительные элементы, стоящие перед минимальным положительным элементом, переслать в «хвост» списка.
5. В списке $A=(a_1, a_2, ..., a_n)$ все положительные элементы, начиная со второго положительного, отправить в хвост списка.
6. В одномерном списке $A=(a_1, a_2, ..., a_n)$ группу элементов, содержащую наибольшее число подряд идущих отрицательных элементов, переписать в «хвост» списка.
7. В одномерном списке $A=(a_1, a_2, ..., a_n)$ все отрицательные элементы, имеющие нечетный порядковый номер, отправить в «хвост» списка, т. е. поместить на место последних элементов.
8. В одномерном списке $A=(a_1, a_2, ..., a_n)$ все положительные элементы, имеющие четный порядковый номер, переписать в начало списка.
9. В одномерном списке $A=(a_1, a_2, ..., a_n)$ отрицательные элементы, имеющие четный порядковый номер, переписать в начало списка.
10. В одномерном списке $A=(a_1, a_2, ..., a_n)$ группу, содержащую наибольшее число подряд идущих положительных элементов, переписать в хвост списка.

Задание 3. Варианты заданий на удаление элементов списка. Элементы списка вводить с помощью генератора случайный чисел

1. В списке $A=(a_1, a_2, ..., a_n)$ удалить последнюю группу положительных элементов. Группой называется подряд идущие элементы одного знака, число которых больше или равно 2.
2. В списке $A=(a_1, a_2, ..., a_n)$ удалить все подряд идущие отрицательные элементы, идущие вслед за минимальным элементом списка.
3. В списке $A=(a_1, a_2, ..., a_n)$ удалить все отрицательные элементы, стоящие перед минимальным элементом списка.
4. В списке $A=(a_1, a_2, ..., a_n)$ удалить все элементы, меньшие, чем элемент списка, расположенный слева от максимального.
5. В списке $A=(a_1, a_2, ..., a_n)$ удалить все элементы, стоящие между минимальным положительным и максимальным отрицательным элементами.
6. В списке $A=(a_1, a_2, ..., a_n)$ удалить все положительные элементы, имеющие четный порядковый номер, идущие после минимального элемента списка.
7. В одномерном списке $A=(a_1, a_2, ..., a_n)$ удалить все отрицательные элементы, расположенные между положительными.
8. В одномерном списке $A=(a_1, a_2, ..., a_n)$ удалить все равные элементы, оставив только один из данных групп равных.
9. В одномерном списке $A=(a_1, a_2, ..., a_n)$ удалить все цепочки отрицательных элементов, расположенные между положительными.

10. В списке A=(a_1, a_2, ..., a_n) удалить все элементы, стоящие между минимальным положительным и максимальным отрицательным элементами.

Задание 4.

Решето Эратосфена — алгоритм нахождения всех простых чисел до некоторого целого числа n, который приписывают древнегреческому математику Эратосфену Киренскому. Как и во многих случаях, здесь название алгоритма говорит о принципе его работы, то есть решето подразумевает фильтрацию, в данном случае фильтрацию всех чисел за исключением простых. По мере прохождения списка нужные числа остаются, а ненужные (они называются составными) исключаются.

Для нахождения всех простых чисел не больше заданного числа n, следуя методу Эратосфена, нужно выполнить следующие шаги:

1. Выписать подряд все целые числа от двух до n (2, 3, 4, …, n).
2. Пусть переменная p изначально равна двум — первому простому числу.
3. Зачеркнуть в списке числа от 2p до n считая шагами по p (это будут числа кратные p: 2p, 3p, 4p, …).
4. Найти первое незачёркнутое число в списке, большее чем p, и присвоить значению переменной p это число.
5. Повторять шаги 3 и 4, пока возможно.

Теперь все незачёркнутые числа в списке — это все простые числа от 2 до n.

Требуется:

а) вычислить все числа от 1 до n, кратные 7;

б) вычислить все простые числа от 1 до n.

Для выполнения задания следует использовать списки и множества.

Задание 5. Варианты заданий на матрицы.

1. Ввести с помощью генератора случайных чисел вещественную матрицу размерности *n*×*m* (заданы константами). Найти максимальный и минимальный элементы и их индексы. Вывести на экран найденные элементы, их индексы.
2. Ввести с помощью генератора случайных чисел вещественную матрицу размерности *n*×*m* (заданы константами). Заменить все отрицательные числа их абсолютным значением. Вывести на экран полученную матрицу.
3. Ввести с помощью генератора случайных чисел вещественную матрицу размерности *n*×*m* (заданы константами). Вычислить среднее арифметическое для каждого столбца. Вывести исходную матрицу и полученные значения на экран.
4. Ввести с помощью генератора случайных чисел вещественную матрицу размерности *n*×*m* (заданы константами). Заменить все положительные числа их квадратом. Вывести на экран полученную матрицу.
5. Ввести с помощью генератора случайных чисел целочисленную матрицу размерности *n*×*m* (заданы константами). Подсчитать количество положительных и отрицательных чисел. Вывести на экран полученные значения.
6. Ввести с помощью генератора случайных чисел целочисленную матрицу размерности *n*×*m* (заданы константами). Найти максимальный элемент среди элементов с нечетной суммой индексов. Вывести на экран найденный элемент, его индексы.
7. Ввести с помощью генератора случайных чисел целочисленную матрицу размерности *n*×*m* (заданы константами). Подсчитать количество четных и нечетных чисел. Вывести на экран полученные значения.

8. Ввести с помощью генератора случайных чисел вещественную матрицу размерности $n \times m$ (заданы константами). Вычислить среднее арифметическое для каждой строки. Вывести исходную матрицу и полученные значения на экран.
9. Ввести с помощью генератора случайных чисел вещественную матрицу размерности $n \times n$ (задана константой). Подсчитать количество нулевых элементов, расположенных на главной и побочной диагоналях.
10. Ввести с помощью генератора случайных чисел целочисленную матрицу размерности $n \times m$ (заданы константами). Найти элемент кратный заданному числу. Вывести на экран найденный элемент, его индексы, степень кратности.

Задание 6. Поиск наибольшего числа в трёхмерном массиве.

Пусть массив *T* имеет размерность 3x5x7. Найти наибольшее содержащееся в нем число и вывести его и его индексы на экран.