

Пройдено

Позже

Типы данных

Числа. 2.078

Пройдено

Позже

Целые числа (int)

Числа в Python ничем не отличаются от обычных чисел. Они поддерживают набор самых обычных математических операций:

$x + y$		Сложение
$x - y$	Вычитание	
$x * y$	Умножение	

x / y	Деление
x // y	Получение целой части от деления
x % y	Остаток от деления
-x	Смена знака числа
abs(x)	Модуль числа
divmod(x, y)	Пара (x // y, x % y)
x ** y	Возведение в степень
pow(x, y[, z])	xy по модулю (если модуль задан)

Например

```

1  # Попробуем поработать с числами
2  >>> 255 + 34
3  289
4  >>> 5 * 2
5  10
6  >>> 20 / 3
7  6.666666666666667
8  >>> 20 // 3
9  6
10 >>> 20 % 3

```

```
11 2
12 >>> 3 ** 4
13 81
14 >>> pow(3, 4)
15 81
16 >>> pow(3, 4, 27)
17 0
18 >>> 3 ** 150
19 369988485035126972924700782451696644186473100389722973815184405301748249
20
21
```

Еще пример

```
1 >>> a = int('19') # Переводим строку в число
2 >>> b = int('19.5') # Строка не является целым числом
3 Traceback (most recent call last):
4   File "", line 1, in
5   ValueError: invalid literal for int() with base 10: '19.5'
6 >>> c = int(19.5) # Применённая к числу с плавающей точкой, отсекает дробную часть
7 >>> print(a, c)
8 19 19
9 >>> bin(19)
10 '0b10011'
11 >>> oct(19)
12 '0o23'
13 >>> hex(19)
14 '0x13'
15 >>> 0b10011 # Так тоже можно записывать числовые константы
16 19
17 >>> int('10011', 2)
18 19
```

```
19 >>> int('0b10011', 2)
20 19
21
22
```

Вещественные числа (float)

```
1
2 # Вещественные числа поддерживают те же операции, что и целые.
3 #
4 # Простенькие примеры работы с числами:
5 >>> c = 150
6 >>> d = 12.9
7 >>> c + d
8 162.9
9 >>> p = abs(d - c) # Модуль числа
10 >>> print(p)
11 137.1
12 >>> round(p) # Округление
13 137
14
15
```

Ложка дегтя

```
1
2 # Python упрощает работу с числами, снимая с программиста заботу о размерности.
3 # Однако из-за этого программист сталкивается с другой проблемой - неточность
4 # вещественных чисел:
5
6 >>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
7 0.9999999999999999
8
9 # Для высокой точности следует использовать другие объекты (например,
10 # decimal и fraction).
11
12 # Также вещественные числа не поддерживают длинную арифметику:
13
14 >>> a = 3 ** 1000
15 >>> a + 0.1
16 Traceback (most recent call last):
17   File "", line 1, in
18   OverflowError: int too large to convert to float
19
20
```

Еще про работу с числами

Помимо стандартных выражений для работы с **числами**, в составе **Python** есть несколько полезных модулей. Модуль **math** предоставляет более сложные математические функции.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

Модуль **random** реализует генератор случайных чисел и функции случайного выбора.

```
>>> import random
>>> random.random()
0.15651968855132303
```

Комплексные числа (complex)

```
1
2 # В Python встроены также и комплексные числа:
3 >>> x = complex(1, 2)
4 >>> print(x)
5 (1+2j)
6 >>> y = complex(3, 4)
7 >>> print(y)
8 (3+4j)
9 >>> z = x + y
10 >>> print(x)
11 (1+2j)
12 >>> print(z)
13 (4+6j)
14 >>> z = x * y
15 >>> print(z)
16 (-5+10j)
17 >>> z = x / y
18 >>> print(z)
19 (0.44+0.08j)
20 >>> print(x.conjugate()) # Сопряжённое число
21 (1-2j)
22 >>> print(x.imag) # Мнимая часть
23 2.0
24 >>> print(x.real) # Действительная часть
```

```
25 1.0
26 >>> print(x > y) # Комплексные числа нельзя сравнить
27 Traceback (most recent call last):
28   File "", line 1, in
29   TypeError: unorderable types: complex() > complex()
30 >>> print(x == y) # Но можно проверить на равенство
31 False
```

Ложка дегтя 2

```
1
2 # Округлением чисел в Python занимается встроенная функция round.
3 # round(number[, ndigits]) - округляет число number до ndigits знаков после запятой
4 # (по умолчанию, до нуля знаков, то есть, до ближайшего целого)
5 >>> round(1.5)
6 2
7 >>> round(2.5)
8 2
9 >>> round(2.65, 1)
10 2.6
11 >>> round(2.75, 1)
12 2.8
13 # Используется "Банковское округление", то есть округление к ближайшему чётному.
14
15 # На практике это оказывается не так уж и важно, например:
16 >>> round(2.85, 1)
17 2.9
18 # Из-за проблем с точностью чисел с плавающей точкой это число чуть больше,
19 # чем 2.85, а потому округляется до 2.9.
20 >>> from fractions import Fraction
21 >>> a = Fraction(2.85)
22 >>> b = Fraction('2.85')
23 >>> a == b
24 False
25 >>> a > b
26 True
27
28
```

Системы счисления

int([object], [основание системы счисления])		- преобразование к целому числу в десятичной системе счисления. По умолчанию система счисления десятичная, но можно задать любое основание от 2 до 36 включительно.
bin(x)	- преобразование целого числа в двоичную строку.	
hex(x)	- преобразование целого числа в шестнадцатеричную строку.	
oct(x)	- преобразование целого числа в восьмеричную строку.	

Строки. "Hello!"

Пройдено

Позже

Литералы строк

Строки в апострофах и в кавычках

```

1 s = 'spam'
2 s = "spam"
3

```



```
1 S = 'spam"s'  
2 S = "spam's"  
3
```

Многострочный текст

```
1 >>> c = '''это очень большая  
2 строка, многострочный  
3 блок текста'''  
4  
5 >>> c  
6 'это очень большая\nстрока, многострочный\nблок текста'  
7  
8 >>> print(c)  
9 это очень большая  
10 строка, многострочный  
11 блок текста  
12  
13
```

Служебные символы

Экранированная последовательность	Назначение
<code>\n</code>	Перевод строки
<code>\a</code>	Звонок
<code>\b</code>	Забой
<code>\f</code>	Перевод страницы
<code>\r</code>	Возврат каретки
<code>\t</code>	Горизонтальная табуляция
<code>\v</code>	Вертикальная табуляция
<code>\N{id}</code>	Идентификатор ID базы данных Юникода
<code>\uhhhh</code>	16-битовый символ Юникода в 16-ричном представлении
<code>\Uhhhh...</code>	32-битовый символ Юникода в 32-ричном представлении
<code>\xhh</code>	16-ричное значение символа
<code>\ooo</code>	8-ричное значение символа
<code>\0</code>	Символ Null (не является признаком конца строки)

"Сырые" строки

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

```
1 s = r'C:\newt.txt'
2
```

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

```
1 S = r'\n\n\'[:-1]
2 S = r'\n\n' + '\\\'
3 S = '\\n\n\'
4
```

Функции строк

Пройдено

Позже

Базовые операции

```
1 Конкатенация (сложение)
2 >>> S1 = 'spam'
3 >>> S2 = 'eggs'
4 >>> print(S1 + S2)
5 'spameggs'
6 Дублирование строки
7 >>> print('spam' * 3)
8 spamspamspam
9 Длина строки (функция len)
10 >>> len('spam')
11 4
12 Доступ по индексу
13 >>> S = 'spam'
```

```
14 >>> S[0]
15 's'
16 >>> S[2]
17 'a'
18 >>> S[-2] # отсчет идет от конца строки
19 'a'
20
21
```

Извлечение среза

```
1 # Оператор извлечения среза: [X:Y]. X – начало среза, а Y – окончание;
2 # символ с номером Y в срез не входит. По умолчанию первый индекс равен 0,
3 # а второй - длине строки.
4 >>> s = 'spameggs'
5 >>> s[3:5]
6 'me'
7 >>> s[2:-2]
8 'ameg'
9 >>> s[:6]
10 'spameg'
11 >>> s[1:]
12 'pameggs'
13 >>> s[:]
14 'spameggs'
15 # Кроме того, можно задать шаг, с которым нужно извлекать срез.
16 >>> s[::-1]
17 'sggemaps'
18 >>> s[3:5:-1]
19 ''
20 >>> s[2::-2]
21 'aeg'
```

22
23**Особенность**

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.

```
1 >>> s = 'spam'
2 >>> s[1] = 'b'
3 Traceback (most recent call last):
4   File "", line 1, in
5     s[1] = 'b'
6   TypeError: 'str' object does not support item assignment
7 >>> s = s[0] + 'b' + s[2:]
8 >>> s
9 'sbam'
10
```

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
str in S	Проверка на вхождение подстроки в строку
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,	

Задание

Следующим образом можно получить текущее время:

```

1 from datetime import datetime
2 current = datetime.now()
3 hour = current.hour
4 minute = current.minute
5

```

Напишите простое приветствие пользователя на ресурсе, вставив в него:

- его имя
- последние его действия

Используйте для этого метода **.format**

Попробуйте то же самое реализовать с помощью простого форматирования: **s % (name, x)**

Списки. [1, 2, 3]

Пройдено

Позже

Создание

Можно обработать любой итерируемый объект встроенной функцией **list**

```
1 >>> list('список')
2 ['с', 'п', 'и', 'с', 'о', 'к']
3
```

Список можно создать и при помощи литерала:

```
1 >>> s = [] # Пустой список
2 >>> l = ['s', 'p', ['isok'], 2]
3 >>> s
4 []
5 >>> l
```

```
6  ['s', 'p', ['isok'], 2]  
7
```

И еще один способ создать список - это **генераторы списков**.

```
1  >>> c = [c * 3 for c in 'list']  
2  >>> c  
3  ['lll', 'iii', 'sss', 'ttt']  
4
```

Пример посложнее

Возможна и более сложная конструкция генератора списков:

```
1  >>> c = [c * 3 for c in 'list' if c != 'i']  
2  >>> c  
3  ['lll', 'sss', 'ttt']  
4  
5  >>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']  
6  >>> c  
7  ['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']  
8
```


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
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 от start до end со значением x
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 (новое в python 3.3)
list.clear()	Очищает список (новое в python 3.3)

Особенность

Методы списков, в отличие от строковых методов, изменяют сам список, потому результат выполнения не нужно записывать в эту переменную.

```

1  >>> l = [1, 2, 3, 5, 7]
2  >>> l.sort()
3  >>> l
4  [1, 2, 3, 5, 7]
5  >>> l = l.sort()
6  >>> print(l)
7  None
8

```

Примеры

```
1 >>> a = [66.25, 333, 333, 1, 1234.5]
2 >>> print(a.count(333), a.count(66.25), a.count('x'))
3 2 1 0
4
5 >>> a.insert(2, -1)
6 >>> a.append(333)
7 >>> a
8 [66.25, 333, -1, 333, 1, 1234.5, 333]
9
10 >>> a.index(333)
11 1
12
13 >>> a.remove(333)
14 >>> a
15 [66.25, -1, 333, 1, 1234.5, 333]
16
17 >>> a.reverse()
18 >>> a
19 [333, 1234.5, 1, 333, -1, 66.25]
20
21 >>> a.sort()
22 >>> a
23 [-1, 1, 66.25, 333, 333, 1234.5]
24
```

Индексы

Как и в других языках программирования, взятие по индексу:

```
1 >>> a = [1, 3, 8, 7]
```

```
2 >>> a[0]
3 1
4 >>> a[3]
5 7
6 >>> a[4]
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   IndexError: list index out of range
10
```

Нумерация элементов начинается с 0. При попытке доступа к несуществующему индексу возникает исключение **IndexError**.

Отрицательные индексы означают нумерацию с конца:

```
1 >>> a = [1, 3, 8, 7]
2 >>> a[-1]
3 7
4 >>> a[-4]
5 1
6 >>> a[-5]
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   IndexError: list index out of range
10
```

Срезы

```
1 >>> a = [1, 3, 8, 7]
2 >>> a[:]          # дублирует весь список
3 [1, 3, 8, 7]
4
```

```
5 >>> a[1:]           # берет срез, начиная с 1 элемента, до конца
6 [3, 8, 7]
7
8 >>> a[:3]           # берет срез с начала до 3 элемента, не включительно
9 [1, 3, 8]
10
11 >>> a[::2]          # дублирует весь список с шагом 2
12 [1, 8]
13
14 >>> a = [1, 3, 8, 7]
15 >>> a[::-1]         # Также все эти параметры могут быть и отрицательными:
16 [7, 8, 3, 1]
17
18 >>> a[:-2]
19 [1, 3]
20
21 >>> a[-2::-1]
22 [8, 3, 1]
23
24 >>> a[1:4:-1]
25 []                 # так как START < STOP, а STEP < 0
26
```

Еще раз

Еще раз: срез с индексами за пределом списка выдаст пустой список:

```
1 >>> a = [1, 3, 8, 7]
2 >>> a[10:20]
3 []
4
```

А теперь **волшебство**:

```
1 >>> a = [1, 3, 8, 7]
2 >>> a[1:3] = [0, 0, 0]      # Таким образом можно изменить сразу несколько элементов списка
3 >>> a
4 [1, 0, 0, 0, 7]
5
6 >>> del a[:-3]              # А Так можно удалить сразу несколько элементов
7 >>> a
8 [0, 0, 7]
9
```

Задание

Делаем программу **"students_list.py"**.

1. Она должна сделать следующее:

- Создаем список.
- Заполняем его нашими именами (+фамилиями).

2. Выводим имя одного студента на экран:

- Получаем индекс через функцию **input()**.
- Выводим на экран студента по этому индексу.

3. Выводим на экран имена нескольких студентов:

- Получаем через **input()** начало и конец среза.
- Выводим на экран студентов из такого среза.

4. Находим количество студентов, в именах которых есть буква **"p"**.

5. Находим группы студентов с одинаковыми именами и создаем списки этих групп.

Кортежи. (1, 2, 3)

Пройдено

Позже

Зачем?

Кортеж, по сути - неизменяемый список.
Зачем нужны кортежи, если есть списки?

Защита от дурака.
Меньший размер.

```
1 >>> a = (1, 2, 3, 4, 5, 6)
2 >>> b = [1, 2, 3, 4, 5, 6]
3
4 >>> a.__sizeof__()
5 36
6
7 >>> b.__sizeof__()
8 44
9
10 # Возможность использовать кортежи в качестве ключей словаря:
11 >>> d = {(1, 1, 1) : 1}
12 >>> d
13 {(1, 1, 1): 1}
14 >>> d = {[1, 1, 1] : 1}
15 Traceback (most recent call last):
16   File "", line 1, in
17     d = {[1, 1, 1] : 1}
18 TypeError: unhashable type: 'list'
19
```

Как работать с кортежами?

Создаем пустой кортеж:

```
1 >>> a = tuple() #С помощью встроенной функции tuple()
2 >>> a
3 ()
4 >>> a = () #С помощью литерала кортежа
5 >>> a
6 ()
7
```

Создаем кортеж из одного элемента:

```
1 >>> a = ('s')
2 >>> a
3 's' # Стоп. Получилась строка! А как же быть??
4
```

```
1 >>> a = ('s', )
2 >>> a
3 ('s',) # Получилось =)
4
```

Продолжаем

Создаем кортеж из нескольких элемента:

```
1 >>> a = ('s', 7, 12, ["some string", 33], (11, 5))
2 >>> a
3 ('s', 7, 12, ["some string", 33], (11, 5))
4
```

Создать кортеж можно из итерируемого объекта с помощью функции **tuple()**

```
1 >>> a = tuple('hello, world!')
2 >>> a
3 ('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')
4
```

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:

```
1 a, b = b, a
2
```

Задание

- Вспомним задание по спискам.
- Изменим его реализацию таким образом, чтобы максимально заменить использование списков на пользование кортежами.
- Удалось ли полностью отказаться от списков?
- Какие возможности списков поддерживаются и в кортежах тоже?

Словари

Пройдено

Позже

Создание

Словари в Python - **неупорядоченные коллекции** произвольных объектов с доступом по ключу.

Их иногда ещё называют **ассоциативными массивами** или **хеш-таблицами**.

```
1 >>> d = {}
2 >>> d
3 {}
4 >>> d = {'dict': 1, 'dictionary': 2}
5 >>> d
6 {'dict': 1, 'dictionary': 2}
7
```

С помощью функции **dict**:

```
1 >>> d = dict(short='dict', long='dictionary')
2 >>> d
3 {'short': 'dict', 'long': 'dictionary'}
4 >>> d = dict([(1, 1), (2, 4)])
5 >>> d
6 {1: 1, 2: 4}
7
```

Продолжаем

С помощью метода **fromkeys**:

```
1 >>> d = dict.fromkeys(['a', 'b'])
2 >>> d
3 {'a': None, 'b': None}
4 >>> d = dict.fromkeys(['a', 'b'], 100)
5 >>> d
6 {'a': 100, 'b': 100}
7
```

С помощью генераторов словарей:

```
1 >>> d = {a: a ** 2 for a in range(7)} # !!! ЭТО ВОЗМОЖНОСТЬ python3
2 >>> d
3 {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
4
```

Пробуем

```
1 >>> d = {1: 2, 2: 4, 3: 9}
2 >>> d[1]
3 2
4
5 >>> d[4] = 4 ** 2
6 >>> d
7 {1: 2, 2: 4, 3: 9, 4: 16} # присвоение по новому ключу расширяет словарь
8
9 >>> d[3] = 'Hi'
```

```

10 >>> d
11 {1: 2, 2: 4, 3: 'Hi', 4: 16} # присвоение по существующему ключу перезаписывает его
12
13 >>> d['1'] # попытка извлечения несуществующего ключа порождает исключение
14 Traceback (most recent call last):
15   File "", line 1, in
16     d['1']
17   KeyError: '1'
18

```

Методы словарей

dict.clear()	очищает словарь.
dict.copy()	возвращает копию словаря.
classmethod dict.fromkeys(seq[, value])	создает словарь с ключами из seq и значением value (по умолчанию None).
dict.get(key[, default])	возвращает значение ключа, но если его нет, не бросает исключение, а возвращает default (по умолчанию None).
dict.items()	возвращает пары (ключ, значение).
dict.keys()	возвращает ключи в словаре.
dict.pop(key[, default])	удаляет ключ и возвращает значение. Если ключа нет, возвращает default (по умолчанию бросает исключение).
dict.popitem()	удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError . Помните, что словари неупорядочены.

dict.setdefault(key[, default])	возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ с значением default (по умолчанию None).
dict.update([other])	обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает None (не новый словарь!).
dict.values()	возвращает значения в словаре.

Задание

- Вспоминаем задание по спискам.
- Теперь вместо списка используем словарь.
- Добавляем в словарь с именами возраст.
- При выводе каждого имени выводить еще возраст студента.

Множества

Пройдено

Позже

Создание

```

1  >>> a = set()           # Пустое множество через функцию set
2  >>> a
3  set()
4
5  >>> a = set('hello')
6  >>> a
7  {'h', 'o', 'l', 'e'}
8
9  >>> a = {'a', 'b', 'c', 'd'}
10 >>> a
11 {'b', 'c', 'a', 'd'}
12
13 >>> a = {i ** 2 for i in range(10)} # генератор множеств

```

```
14 >>> a
15 {0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
16
17 >>> a = {} # А так нельзя!
18 >>> type(a)
19
20 # множества имеет тот же литерал, что и словарь, но пустое множество
21 # с помощью литерала создать нельзя.
22 <class 'dict'>
23
```

Удобно

Множества удобно использовать для удаления повторяющихся элементов:

```
1 >>> words = ['hello', 'daddy', 'hello', 'mum']
2 >>> set(words)
3 {'hello', 'daddy', 'mum'}
4
```

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

```
1 len(s) # число элементов в множестве (размер множества).
2 x in s # принадлежит ли x множеству s.
3 set.isdisjoint(other) # истина, если set и other не имеют общих элементов.
4 set == other # все элементы set принадлежат other, все элементы other принадлежат set.
```

```
5 set.issubset(other) или set <= other # все элементы set принадлежат other.
6 set.issuperset(other) или set >= other # аналогично.
7 set.union(other, ...) или set | other | ... # объединение нескольких множеств.
8 set.intersection(other, ...) или set & other & ... # пересечение.
9 set.difference(other, ...) или set - other - ... # множество из всех элементов set,
10   # не принадлежащие ни одному из other.
11 set.symmetric_difference(other); set ^ other # множество из элементов, встречающихся
12   # в одном множестве, но не встречающиеся в обоих.
13 set.copy() # копия множества.
14
```

Операции, изменяющие множество

```
1 set.update(other, ...); set |= other | ... # объединение.
2
3 set.intersection_update(other, ...); set &= other & ... # пересечение.
4
5 set.difference_update(other, ...); set -= other | ... # вычитание.
6
7 set.symmetric_difference_update(other); set ^= other
8
9 set.add(elem) # добавляет элемент в множество.
10
11 set.remove(elem) # удаляет элемент из множества. KeyError, если такого элемента
12   # не существует.
13
14 set.discard(elem) # удаляет элемент, если он находится в множестве.
15
16 set.pop() # удаляет первый элемент из множества. Так как множества не упорядочены,
17   # нельзя точно сказать, какой элемент будет первым.
18
19 set.clear() # очистка множества.
20
```

frozenset

```
1  # Единственное отличие set от frozenset заключается в том, что
2  # set - изменяемый тип данных, а frozenset - нет.
3
4  >>> a = set('qwerty')
5  >>> b = frozenset('qwerty')
6  >>> a == b
7  True
8  >>> type(a - b)
9
10 >>> type(a | b)
11
12 >>> a.add(1)
13 >>> b.add(1)
14 Traceback (most recent call last):
15   File "", line 1, in
16     b.add(1)
17 AttributeError: 'frozenset' object has no attribute 'add'
18
```


Задание

- Вспоминаем задание по спискам.
- Выводим список имен без повторений.
- Для чего еще вы бы использовали множества?

Пробуем

Пройдено

Позже

Для домашней работы

Сохранить объект в файл можно так:

```
1 >>> import pickle
2 >>> data = {
3 ...     'a': [1, 2.0, 3, 4+6j],
4 ...     'b': ("character string", b"byte string"),
5 ...     'c': {None, True, False}
6 ... }
7 >>>
8 >>> with open('data.pickle', 'wb') as f:
9 ...     pickle.dump(data, f)
10 ...
11 >>> with open('data.pickle', 'rb') as f:
12 ...     data_new = pickle.load(f)
13
```

1. Сделайте простую базу данных:

- Пользователь вводит команду: **ввести, вывести**
- Ввести - пользователь вводит марку автомобиля и его мощность. Необходимо

проверить, что марка состоит только из букв латинского или русского алфавитов.
Мощность только из цифр.

- Вывести - выводятся все автомобили - по алфавиту. Сортировку сделать сначала стандартным методом. Затем написать свою версию сортировки циклами.

2. Реализовать поиск/фильтрацию в базе данных - то есть вывод по условию.

- По мощности - конкретное число, больше, меньше, в промежутке.
- По вхождению слова в название.
- По полному соответствию слова.