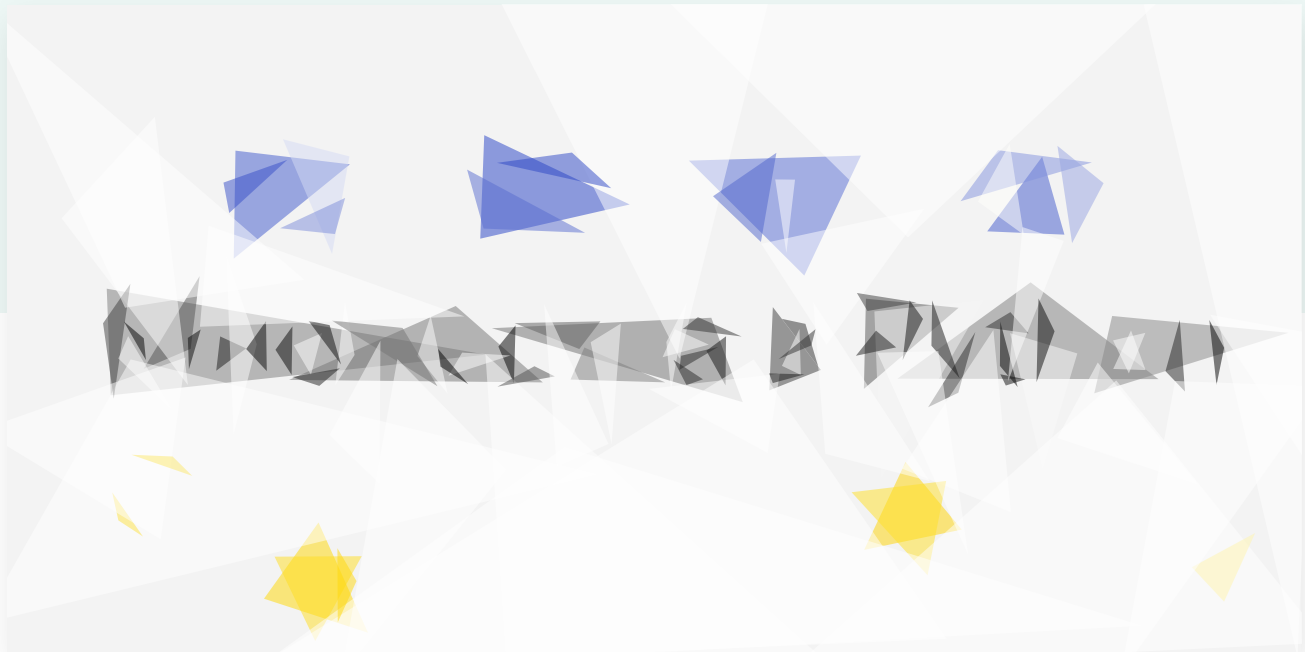




— ПИТОНЧИК



06.06.2020 Александр Зайков

Основы

## Множества в Python (set, frozenset)

### Содержание:

- Назначение в Python
- Особенности set
- Работа с set-ами
  - Создание
  - Пересечение
  - Добавление элемента
  - Удаление и очистка
  - Перебор элементов
  - Принадлежность объекта set-у
  - Сортировка множеств

## Длина множества

- Операции на множествах

Объединение

Пересечение

Разность множеств

Симметрическая разность

`isdisjoint()`

`issubset()`

`issuperset()`

`update()`

`intersection_update()`

`difference_update()`

`symmetric_difference_update()`

- Свойства методов и операторов

- Преобразования

Конвертация строки во множество


Конвертация списка во множество

- Frozenset

**Множество** – интуитивно понятный математический термин, который часто используется в обыденной речи и означает набор или совокупность неких элементов, что обладают каким-то общим свойством.

Не слишком строгое определение множества, однако, с ним возникали проблемы даже у великих математиков.

В широком смысле, элементами множеств могут быть даже нематериальные вещи: чётные числа, несданные задачи по термодинамике, алгоритмы сортировки, любимые фильмы Юлии и Алексея и даже мысли об эклерах.

 Возьмите в руки кота. Взяли? Хорошо. Теперь множество котов в ваших руках насчитывает ровно один мурлыкающий элемент. Если же пушистику вдруг не понравится, что вы его тискаете, и он выскочит из рук, то элементов внутри множества не останется. Множество, в котором нет ни одного элемента, называется пустым. Но что же там в Python?

# Назначение в Python

Множества (set) в питоне появились не сразу, и здесь они представлены как неупорядоченные коллекции уникальных и неизменяемых объектов.

Коллекции, которые не являются ни последовательностями (как списки), ни отображениями (как словари). Хотя с последними у множеств много общего.

Можно сказать, что set напоминает словарь, в котором ключи не имеют соответствующих им значений

Множества:

- Дают возможность быстро удалять дубликаты, поскольку, по определению, могут содержать только уникальные элементы;
- Позволяют, в отличие от других коллекций, выполнять над собой ряд математических операций, таких как объединение, пересечение и разность множеств;

Пример set-ов в Python:

```
# множество натуральных чисел от 1 до 10
natural_num_set = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

# множество персонажей Братства Кольца
the_fellowship_of_the_ring_set = {'Гэндальф', 'Арагорн', 'Фродо', 'Сэм', 'Б

# множество приближений math.sqrt(2)
sqrt_approximation_set = {1.4142135623, 1.414213562, 1.41421356, 1.4142135,

# множество результатов какого-то голосования
vote_result_set = {('P', 76.69), ('G', 11.77), ('J', 5.65), ('S', 1.68), ('
```

## Особенности set

Одно из основных свойств множеств заключается в уникальности каждого из их элементов. Посмотрим, что получится, если сформировать set из строки с заведомо повторяющимися символами:

```
strange_app = set('TikTok')
print(strange_app)
```

```
> {'o', 'T', 'i', 'k'}
```

Из результата были удалены дублирующиеся в слове 'TikTok' символы. Так множества в очередной раз доказали, что содержат в себе только уникальные элементы.

☞ Немаловажным является и тот факт, что при литеральном объявлении, итерируемые объекты сохраняют свою структуру.

```
pangram = {'съешь же ещё этих мягких французских булок, да выпей чаю'}
print(pangram)
> {'съешь же ещё этих мягких французских булок, да выпей чаю'}
```

Для сравнения:

```
pangram_second = set('съешь же ещё этих мягких французских булок, да выпей чаю')
print(pangram_second)
# попытка чаю с функцией set(), к сожалению, не выйдем
> {'щ', 'ь', 'э', 'н', 'з', 'с', 'м', ' ', 'р', 'о', 'ю', 'ш', 'ё', 'к', 'у'}
```

Отдельное python множество может включать в себя объекты разных типов:

```
we_are_the_py_objects = {None, True, 42, 3.14, 'Force', ('Dark Side', 'Light Side')}
print(we_are_the_py_objects)
> {True, 3.14, ('Dark Side', 'Light Side'), 42, 'Force', None}
```

Здесь нет никакого противоречия с математической дефиницией, так как все составляющие `we_are_the_py_objects` имеют вполне конкретное общее свойство, являясь объектами языка Питон.

Но не стоит забывать и внутреннее определение set-ов. Важно помнить, что list-ы и dict-ы не подходят на роль элементов множества, из-за своей изменяемой природы.

```
glados = [['Great cake']]
print(glados)
```

```
> Traceback (most recent call last):
  glados = {[ 'Great cake' ]}
TypeError: unhashable type: 'list'
```

Функция `set()`, тем не менее, корректно обрабатывает случаи, когда ей на вход подаются списки или словари.

```
# словарь будет преобразован во множество его ключей, значения отбрасываются.
some_dict = {'key_one': 'val_one', 'key_two': 'val_two'}
some_set = set(some_dict)
print(some_set)

> {'key_one', 'key_two'}

# элементы списка преобразуются в элементы множества, дубликаты удаляются
card_suit = ['heart', 'diamond', 'club', 'spade', 'spade']
suit_set = set(card_suit)
print(suit_set)

> {'club', 'diamond', 'spade', 'heart'}
```

Однако в списках не должно быть вложенных изменяемых элементов.

```
tricky_list = [{'jocker': 'black'}, {'jocker': 'red'}]
sad_set = set(tricky_list)
print(sad_set)

> Traceback (most recent call last):
  sad_set = set(tricky_list)
TypeError: unhashable type: 'dict'
```

## Работа с set-ами

### Создание

set= {`[]`}

Объявим Python-множество `S`. Существует два способа это сделать:

**Способ №1.** Воспользовавшись литералом:

```
S = {'1', '2', '3'}
print(S)

> {'1', '2', '3'}

print(type(S))

> <class 'set'>
```

**Способ №2.** Применив встроенную функцию `set()`:

Чтобы получить аналогичный результат, необходимо передать итерируемый объект (список, строку или кортеж) в качестве аргумента:

```
# объявим список L
L = ['1', '2', '3']
# и предоставим его в set()
S_2 = set(L)
print(S_2)
# так как set – коллекция неупорядоченная, то результат вывода может отличаться
> {'1', '2', '3'}

print(type(S_2))

> <class 'set'>
```

📖 Замечание: пустое множество создаётся исключительно через `set()`

```
empty_set = set()
print(empty_set)
> set()

print(type(empty_set))
> <class 'set'>
```

Если же сделать так:

```
another_empty_set = {}  
print(another_empty_set)  
> {}  
  
print(type(another_empty_set))  
> <class 'dict'>
```

То получим пустой словарь. А если внутри фигурных скобок поместить пустую строку:

```
maybe_empty_set = {''}  
print(maybe_empty_set)  
> {''}  
  
print(type(maybe_empty_set))  
> <class 'set'>
```

То на выходе увидим множество, состоящее из одного элемента – этой самой пустой строки.

```
# количество элементов множества  
print(len(maybe_empty_set))  
  
> 1
```

Вполне естественно, что пустое множество, при приведении его к логическому типу, тождественно ложно:

```
true_or_false = set()  
print(bool(true_or_false))  
  
> False
```

## Пересечение



В программировании нередко задачи, в которых требуется найти совпадающие элементы двух коллекций. Классическое решение основано на цикле `for`, но нас интересует другое – то, что строится на использовании `set`-ов.

```
# пусть есть два списка, и нужно отыскать и вывести их одинаковые компоненты
unbreakable_diamond = ['Jotaro', 'Josuke', 'Koichi']
golden_wind = ['Jotaro', 'Koichi', 'Giorno']
# & - здесь оператор пересечения
overlap = set(unbreakable_diamond) & set(golden_wind)
print(overlap)

> {'Jotaro', 'Koichi'}
```

## Добавление элемента

`set = {`  `}`

Для добавления нового элемента в существующий набор используем метод `add(x)`.

```
stats = {1.65, 2.33, 5.0}
stats.add(14.7)
print(stats)

> {1.65, 2.33, 5.0, 14.7}
```

Если среди исходных объектов, составляющих `set`, "x" уже был, то ничего не произойдёт, и начальное множество не изменится.

```
big_cats = {'tiger', 'liger', 'lion', 'cheetah', 'leopard', 'cougar'}
big_cats.add('cheetah')
# это жестоко, но второго гепарда не появится
print(big_cats)

> {'cheetah', 'liger', 'cougar', 'lion', 'tiger', 'leopard'}
```

## Удаление и очистка



set = { 

проблем благодаря методу `clear()`.

К пустому не составит никаких

```
set_with_elements = {'i am element', 'me too'}  
print(set_with_elements)
```

```
> {'i am element', 'me too'}
```

```
set_with_elements.clear()  
print(set_with_elements)
```

```
> set()
```

Для удаления одного единственного компонента из набора в Питоне определены аж три способа.

**Способ №1.** Метод `remove()`.

Метод удаляет элемент `elem` из `set`-а. В случае отсутствия `elem` в наборе интерпретатор выбрасывает исключение.

```
point_coord = {('x', 52.4), ('y', -5), ('z', 0.3)}  
print(point_coord)
```

```
> {('y', -5), ('z', 0.3), ('x', 52.4)}
```

```
point_coord.remove(('x', 52.4))  
print(point_coord)
```

```
> {('y', -5), ('z', 0.3)}
```

```
point_coord.remove(('z', 11.8))  
print(point_coord)
```

```
> Traceback (most recent call last):  
  point_coord.remove(('z', 11.8))  
KeyError: ('z', 11.8)
```

**Способ №2.** Метод `discard()`.

Производит предельно схожую с `remove()` операцию с той лишь разницей, что, в случае отсутствия элемента в коллекции, исключение не возникает:

```
triangle_coord = {(0, 4), (3, 0), (-3, 0)}
```

```

print(triangle_coord)

> {(3, 0), (-3, 0), (0, 4)}

triangle_coord.discard((0, 4))
print(triangle_coord)

> {(3, 0), (-3, 0)}

triangle_coord.discard((54, 55))
print(triangle_coord)

> {(3, 0), (-3, 0)}

```

### Способ №3. Метод `pop()` .

Удаляет и возвращает случайный элемент множества:

```

sweets = {'candy', 'gingerbread', 'lollipop', 'candy floss', 'sweets'}
print(sweets)
> {'lollipop', 'candy', 'sweets', 'gingerbread', 'candy floss'}

print(sweets.pop())
> lollipop

print(sweets)
> {'candy', 'sweets', 'gingerbread', 'candy floss'}

```

## Перебор элементов

set = {



Множество, как и любую другую коллекцию, итерируем циклом `for` :



```


iterate_me = {1.1, 1.2, 1.3, 1.4, 1.5}
for num in iterate_me:
    print(num)

>
1.1
1.4
1.3

```

## Принадлежность объекта set-у

set = {   }

“A”? 


Оператор `in` даёт возможность проверить наличие элемента в наборе:

```
berry_club = {'Tomato', 'Currant', 'Sea buckthorn', 'Grape', 'Barberry'}
print('Tomato' in berry_club)
> True

print('Strawberry' in berry_club)
> False
```

## Сортировка множеств

set                      list





{    } => [    ]


Операция сортировки отсутствует для множеств Python по определению. Множество – неупорядоченный набор. Но не нужно расстраиваться. С помощью функции `sorted()`, вы всегда можете получить отсортированный список:

```
some_digits = {1, 55, 34, 2, 12, 14, -4}
print(sorted(some_digits))
> [-4, 1, 2, 12, 14, 34, 55]

cities = {'Москва', 'Калининград', 'Новосибирск', 'Архангельск', 'Белгород'}
print(sorted(cities))
> ['Архангельск', 'Белгород', 'Калининград', 'Москва', 'Новосибирск', 'Хаба
```

## Длина множества

set = {     }

len = 4 

Размеры определенного `set`-а получаем функцией `len()` :

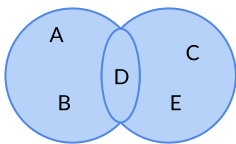
```
eng_alph = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm'}
print(len(eng_alph))
> 26
```

## Операции на множествах

Самое важное в этой теме. Математические теоретико-множественные операции, что не доступны никаким другим коллекциям языка. Поехали.

### Объединение

(оператор `|`, логическое `или` )



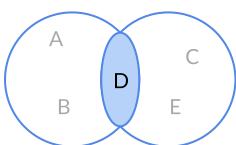
Объединением двух множеств "X" и "Y" является такое третье множество "Z", каждый элемент которого принадлежит либо множеству "X", либо "Y".

```
lang_X = {'C++', 'Perl', 'PHP'}
lang_Y = {'Java', 'C#', 'PHP', 'Python'}
lang_Z = lang_X.union(lang_Y) # или так lang_Z = lang_X | lang_Y
print(lang_Z)

> {'Java', 'C#', 'Python', 'PHP', 'C++', 'Perl'}
```

### Пересечение

(оператор `&`, логическое `и` )



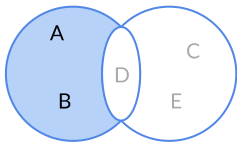
Пересечением двух множеств "A" и "B" является такое третье множество "C", каждый элемент которого принадлежит и множеству "A", и множеству "B".

```
bats_enemies = {'Darkside', 'Joker', 'Bane'}
sups_enemies = {'General Zod', 'Darkside', 'Lobo'}
JL_enemies = bats_enemies.intersection(sups_enemies)
# или так JL_enemies = bats_enemies & sups_enemies
print(JL_enemies)

> {'Darkside'}
```

## Разность множеств

(оператор `-`)

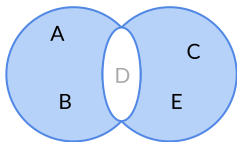


Разностью двух множеств "O" и "P" является такое третье множество "S", каждый элемент которого принадлежит множеству "O" и не принадлежит множеству "P".

```
minuend = {23, 44, 1, 34, 98}
subtrahend = {23, 44, 1, 55, 76}
total = minuend.difference(subtrahend) # или так total = minuend - subtrahend
print(total)
> {34, 98}
```

## Симметрическая разность

(оператор `^`)



Симметрической разностью двух множеств "M" и "N" является такое третье множество "L", каждый элемент которого принадлежит либо множеству "M", либо "N", но не их пересечению.

```
f_set = {11, 'a', 18, 'v', 65, 'g'}
s_set = {11, 'z', 32, 'v', 0, 'g'}
t_set = f_set.symmetric_difference(s_set) # или так t_set = f_set ^ s_set
```

```
print(t.set)
```

Помимо теоретико-множественных операций, в питоне существуют и сугубо утилитарные производные методы.

## isdisjoint()

Метод определяет, есть ли у двух set-ов общие элементы:

```
it = {'green', 'white', 'red'}
ru = {'white', 'blue', 'red'}
ukr = {'blue', 'yellow'}

# вернет False, если множества пересекаются
print(ukr.isdisjoint(it))

> True

# и True, в противном случае
print(ru.isdisjoint(it))

> False
```

В Python нет оператора, который бы соответствовал этому методу.

## issubset()

Показывает, является ли "I" подмножеством "J" (Метод вернет True, если все элементы "I" принадлежат "J"):

```
solar_system = {'Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Pluto'}
first_three_planets = {'Mercury', 'Venus', 'Earth'}
poor_small_guy = {'Pluto'}
emptyness = set()

print(first_three_planets.issubset(solar_system))
# или так first_three_planets <= solar_system
> True

print(poor_small_guy.issubset(solar_system))
# poor_small_guy <= solar_system
> False

# как и в математике, пустое множество есть подмножество любого множества
print(emptyness.issubset(solar_system))
```

```
# emptyness <= solar_system
> True

# также любое множество является подмножеством самого себя
print(poor_small_guy.issubset(poor_small_guy))
# poor_small_guy <= poor_small_guy
> True
```

Оператор `<` определяет, является ли одно множество строгим подмножеством другого. В большинстве ситуаций, он ведёт себя точно так же, как и `<=`, кроме последнего случая. Сравниваемые наборы не должны быть идентичными. Таким образом, для данного оператора не существует соответствующего метода.

```
print(poor_small_guy.issubset(poor_small_guy))
# poor_small_guy < poor_small_guy

> False
```

## issuperset()

Показывает, является ли "F" надмножеством "G":

```
print(solar_system.issuperset(first_three_planets))
# solar_system >= first_three_planets
> True

print(poor_small_guy.issuperset(solar_system))
# poor_small_guy >= solar_system

> False

# в сердечке Плутона лишь пустота...
print(poor_small_guy.issuperset(emptyness))
# poor_small_guy >= emptyness

> True
```

Особенности оператора строгого надмножества `>` идентичны таковым у `<`.

```
print(poor_small_guy > poor_small_guy)
> False
```

И для него в языке Python тоже не существует соответствующего метода.

Квартет методов, которые присваивают `set` -у результат его объединения с другим множеством. "Другое множество" передаётся методу в качестве аргумента.

## update()

Изменяет исходное множество по объединению:

```
dogs_in_first_harness = {'Lessie', 'Bork', 'Spark'}
dogs_in_second_harness = {'Lucky'}
dogs_in_second_harness.update(dogs_in_first_harness)
# или так dogs_in_second_harness |= dogs_in_first_harness
print(dogs_in_second_harness)

> {'Spark', 'Bork', 'Lucky', 'Lessie'}
```

## intersection\_update()

По пересечению:

```
basin_measurements_west = {-8745, -9000, -7990}
basin_measurements_east = {-7990, -6934, -8100}
basin_measurements_west.intersection_update(basin_measurements_east)
# или так basin_measurements_west &= basin_measurements_east
print(basin_measurements_west)

> {-7990}
```

## difference\_update()

По разности:

```
prices_may = {100, 200, 125}
prices_june = {100, 200, 300}
prices_may.difference_update(prices_june) # или так prices_may -= prices_june
print(prices_may)

> {125}
```

## symmetric\_difference\_update()



И, наконец, по симметрической разности:

```
his_bag = {'croissant', 'tea', 'cookies'}
her_bag = {'tea', 'cookies', 'chocolate', 'waffles'}
her_bag.symmetric_difference_update(his_bag)
print(her_bag) # или так her_bag ^= his_bag

> {'croissant', 'chocolate', 'waffles'}
```

## Свойства методов и операторов

Как показано выше, данные операции, за некоторым исключением, выполняются двумя способами: при помощи метода или соответствующего ему оператора (например `union()` и оператор `|`). Главным и основным их различием является то, что метод может принимать в качестве аргумента не только `set`, но и любой итерируемый объект, в то время, как оператор требует в качестве операндов наличие фактических множеств.

```
list_of_years = [2019, 2018, 2017]
set_of_years = {2009, 2010, 2011}
print(set_of_years.union(list_of_years))

> {2017, 2018, 2019, 2009, 2010, 2011}

print(set_of_years | list_of_years)

>
Traceback (most recent call last):>
  print(set_of_years | list_of_years)
TypeError: unsupported operand type(s) for |: 'set' and 'list'
```

Но есть и сходства. Например, важным является то, что некоторые операторы и методы позволяют совершать операции над несколькими сетами сразу:

```
one = set('11111')
two = set('22222')
three = set('33333')
four = set('44444')
five = set('55555')

uni_set_v1 = one.union(two, three, four, five)
print(uni_set_v1)
```

```

> {'2', '1', '5', '3', '4'}

uni_set_v2 = one | two | three | four | five
print(uni_set_v2)

> {'2', '1', '5', '3', '4'}

es1 = {11, 21, 31, 311, 3111}
es2 = {111, 211, 311, 411}
es3 = {1111, 2111, 3111, 4111}
# порядок выполнения операций слева --> направо
print(es1 - es2 - es3)

> {11, 21, 31}
print(es1.difference(es2, es3))
> {11, 21, 31}

```

Тем интереснее, что оператор `^` симметрической разности позволяет использовать несколько наборов, а метод `symmetric_difference()` – нет.

```

tc1 = {10.1, 20.2, 30.3, 40.4, 50.5}
tc2 = {10.1, 20.2, 30.3, 40.4, 500}
tc3 = {1, 50.1, 1000}

print(tc1 ^ tc2 ^ tc3) # вы же помните про порядок операций (слева-направо)

> {1, 1000, 50.1, 50.5, 500}

print(tc1.symmetric_difference(tc2, tc3))

> Traceback (most recent call last):
  print(tc1.symmetric_difference(tc2, tc3))
TypeError: symmetric_difference() takes exactly one argument (2 given)

```

# Преобразования

## Конвертация строки во множество

Чтобы перевести строку во множество, достаточно представить её в виде литерала этого множества.

```

my_string = 'Lorem ipsum dolor sit amet'
sting_to_set = {my_string}

```

```
print(sting_to_set)

> {'Lorem ipsum dolor sit amet'}
```

## Конвертация списка во множество

Со списком подобный трюк не пройдет, но здесь на помощь спешит функция

`set()` :

```
my_list = [2, 4, 8, 16, 32]
list_to_set = set(my_list)
print(list_to_set)

> {32, 2, 4, 8, 16}
```

## Frozenset

Закончим статью описанием такой структуры данных, что максимально близка Python-множествам. Она называется `Frozenset` .

`Frozen set` -ы, по сути, отличаются от обычных лишь тем, что являются неизменяемым типом данных, в то время, как простой `set` возможно изменять.

Приведём пример:

```
ordinary_set = set()
frozen_set = frozenset()

print(ordinary_set == frozen_set)
> True

# на них также определены теоретико-множественные операции
print(type(ordinary_set-frozen_set))
> <class 'set'>

# отличие кроется в неизменяемости
ordinary_set.add(1)
print(ordinary_set)
> {1}

frozen_set.add(1)
> Traceback (most recent call last):
```

```
frozen_set.add(1)
AttributeError: 'frozenset' object has no attribute 'add'
```

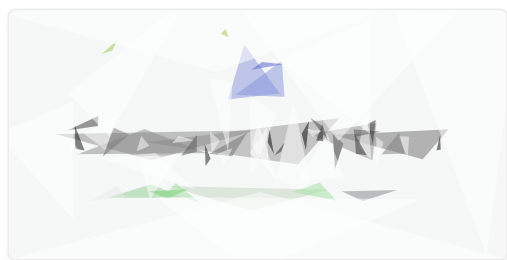
4



Если вам понравилась статья, поделитесь ссылкой на нее



Может понравиться



ОСНОВЫ 06.06.2020

Словари в Python (dict)



ОСНОВЫ 06.06.2020

Кортежи в Python (tuple)



ОСНОВЫ 26

Операторы и в Python

Py

© pythonchik.ru, 2020

[info@pythonchik.ru](mailto:info@pythonchik.ru)