Множества

План урока

1. Объекты типа set и frozenset
2. Операции над множеством
3. Операции над двумя множествами
4. Сравнение множеств
5. Преобразования между списком и множеством

Аннотация

В этом уроке мы обсудим множества Python. Этот тип данных аналогичен математическим множествам, он поддерживает быстрые операции проверки наличия элемента в множестве, добавление и удаление элементов, а также операции объединения, пересечения, вычитания множеств.

Объекты типа set и frozenset

Множество в Python — это объект типа `set`, содержащий *неупорядоченный* набор **различных** элементов. Объекты множеств можно записать следующим образом:

```
animals = {'cat', 'dog', 'fox', 'elephant'}
```

Конфликта с синтаксисом записи словарей не происходит, так как в словаре пары ключ-значение разделяются двоеточием, а во множестве пар нет и все элементы записаны через запятую. Но пустые фигурные скобки уже заняты для создания пустого словаря, поэтому для **создания пустых множеств** обязательно вызывать **функцию `set`**:

```
empty = set()
```

Для элементов множества действует важное ограничение: они должны быть **неизменяемыми**. Как мы знаем, к неизменяемым типам данных относятся числа, строки, кортежи. Например, поместить списки во множество не получится, вместо них следует использовать кортежи:

```
animals_colors = {('cat', 'grey'), ('dog', 'brown'),
                  ('for', 'red'), ('elephant', 'grey')}
```

Так же есть неизменяемые множества, которые имеют тип **`frozenset`**. Неизменяемые множества полностью аналогичны обычным, однако в них отсутствуют операции, которые могут изменить набор элементов после создания.

У множеств есть три коренных отличия от списков:

- Порядок элементов во множестве не определён.
- Множество не может содержать одинаковых элементов.
- Элементы множеств — всегда данные неизменяемого типа.

Выполнение этих трёх свойств позволяет организовать элементы множества в структуру со сложными взаимосвязями, благодаря которым можно быстро проверять наличие элементов во множестве, объединять множества и так далее. Но пока давайте обсудим эти ограничения.

Первые два свойства являются следствиями математического определения множества:

Множество считается заданным, если для любого объекта можно сказать, принадлежит он множеству или нет. То есть множество задаётся некоторым высказыванием, которое может быть истинно (элемент принадлежит множеству) или ложно (не принадлежит).

Отметим для себя два момента:

- Множество не упорядочено. Мы можем узнать, принадлежит элемент множеству или нет. Порядок следования элементов узнать нельзя.
- Нельзя сказать, что элемент входит в множество несколько раз. У нас есть только возможность проверить, есть он во множестве или нет. Поэтому будем считать, что если элемент есть во множестве, то он там только один.

Неизменяемость является дополнительным техническим требованием, позволяющим реализовать быстрые операции с множествами. Дело в том, что элементы множества специальным сложным образом упорядочены и при изменении элемента после добавления во множество его пришлось бы переупорядочивать по-новому, что привело бы к дополнительным временным затратам. Поэтому такие вещи запрещены и элементами множеств могут быть только данные неизменяемых типов.

Операции над множеством

Простейшая операция — **вычисление числа элементов** множества. Для это служит функция `len`:

```
my_set = {'a', 'b', 'c'}
n = len(my_set)  # => 3
```

Далее можно **вывести элементы множества** с помощью функции `print`:

```
my_set = {'a', 'b', 'c'}
print(my_set) # => {'b', 'c', 'a'}
```

В вашем случае порядок может отличаться, так как правило упорядочивания элементов во множестве выбирается случайным образом при запуске интерпретатора Python.

Так же очень часто необходимо обойти все элементы множества в цикле, для этого отлично подходит цикл `for`:

```
my_set = {'a', 'b', 'c'}
for elem in my_set:
    print(elem)
```

такой код выводит:

```
b
a
c
```

Однако, как сказано ранее, в вашем случае порядок может отличаться: заранее он неизвестен. Код для работы с множествами нужно писать таким образом, чтобы он правильно работал при любом порядке обхода. Для этого надо знать два правила:

- Если мы не изменяли множество, то порядок обхода элементов тоже не изменится.
- После изменения множества порядок элементов может поменяться коренным образом.

Чтобы **проверить наличие элемента** во множестве, можно воспользоваться уже знакомым оператором `in`:

```
if elem in my_set:
    print('Элемент во множестве')
else:
    print('Элемент отсутствует')
```

Выражение `elem in my_set` возвращает `True`, если элемент есть во множестве, и `False`, если его нет. Эта операция для множеств выполняется за время, не зависящее от числа элемента во множестве.

Добавить элемент во множество можно с помощью метода `add`:

```
new_elem = 'e'
my_set.add(new_elem)
```

Если элемент, равный `new_elem`, уже существует во множестве, то оно не изменится, поскольку не может содержать одинаковых элементов. Ошибки при этом не произойдёт. Небольшой пример:

```
my_set = set()
my_set.add('a')
my_set.add('b')
my_set.add('a')
print(my_set)
```

Данный код выведет либо `{'a', 'b'}`, либо `{'b', 'a'}`.

С **удалением элемента** сложнее. Для этого есть сразу три метода: `discard` (удалить заданный элемент, если он есть во множестве, и ничего не делать, если его нет), `remove` (удалить заданный элемент, если он есть, и породить ошибку `KeyError`, если нет) и `pop`. Метод `pop` удаляет некоторый элемент из множества и возвращает его как результат. Порядок удаления при этом неизвестен.

```
my_set = {'a', 'b', 'c'}

my_set.discard('a')         # Удалён
my_set.discard('hello')     # Не удалён, ошибки нет

my_set.remove('b')          # Удалён
print(my_set)               # В множестве остался один элемент 'c'
my_set.remove('world')      # Не удалён, ошибка KeyError
```

На первый взгляд, странно, что есть метод `remove`, который увеличивает количество падений вашей программы. Однако, если вы на 100 процентов уверены, что элемент должен быть в множестве, то лучше получить ошибку во время отладки и исправить её, чем тратить время на поиски при неправильной работе программы.

Метод `pop` удаляет из множества случайный элемент и возвращает его значение:

```
my_set = {'a', 'b', 'c'}
print('до удаления:', my_set)

elem = my_set.pop()

print('удалённый элемент:', elem)
print('после удаления:', my_set)
```

Результат работы случаен, например, такой код может вывести следующее:

```
до удаления: {'b', 'a', 'c'}
удалённый элемент: b
после удаления: {'a', 'c'}
```

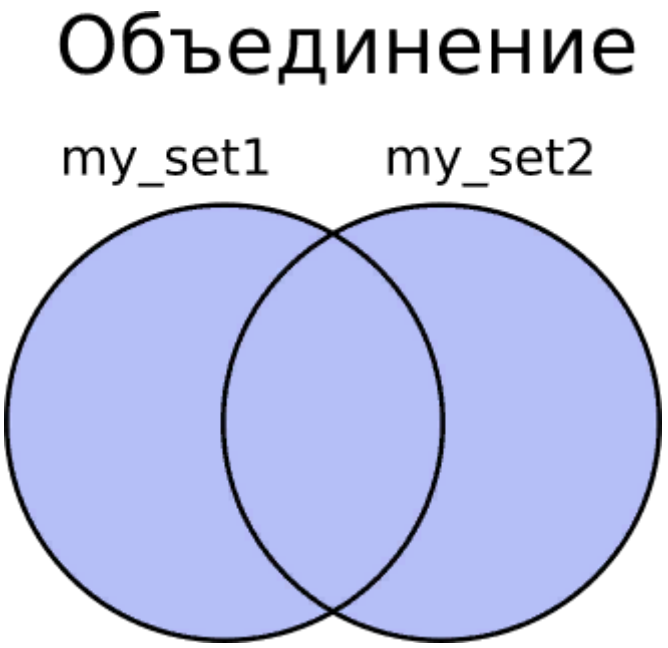
Если попытаться применить `pop` к пустому множеству, произойдёт ошибка `KeyError`.

Очистить множество от всех элементов можно методом `clear`:

```
my_set.clear()
```

Операции над двумя множествами

Есть четыре операции, которые из двух множеств делают новое множество: объединение, пересечение, разность и симметричная разность.

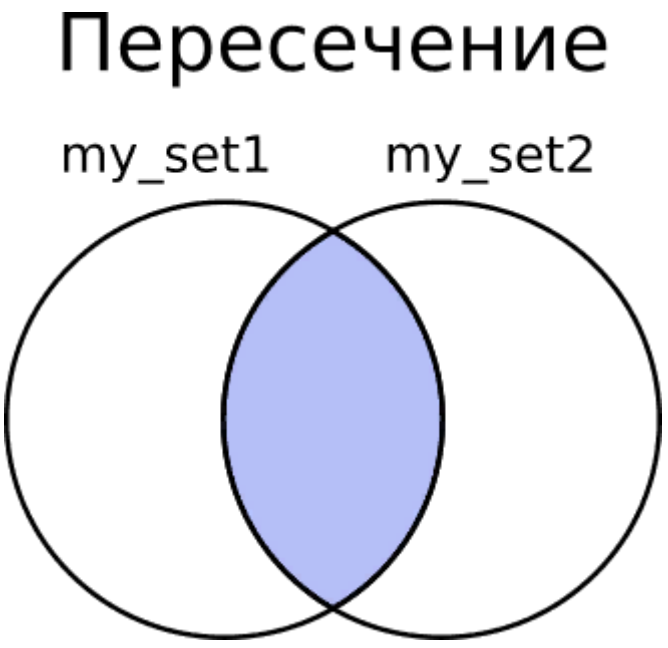


Объединение двух множеств включает в себя все элементы, которые есть хотя бы в одном из них. Для этой операции существует метод `union`:

```
union = my_set1.union(my_set2)
```

Или можно использовать оператор `|`:

```
union = my_set1 | my_set2
```

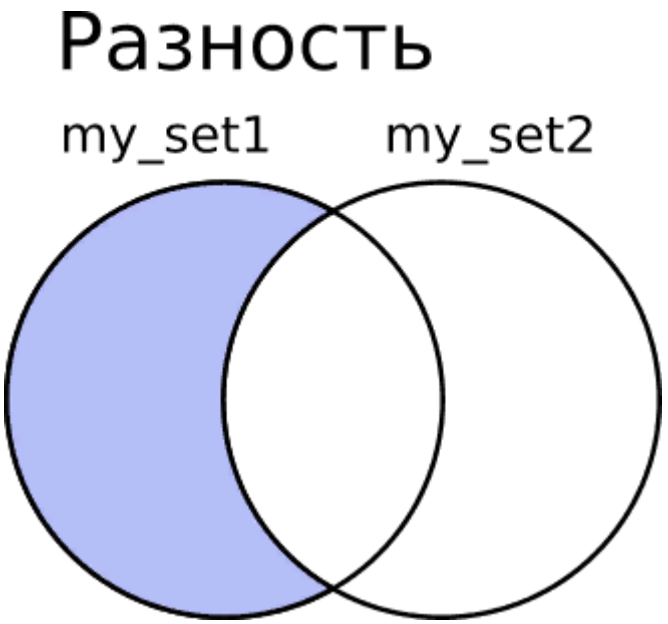


Пересечение двух множеств включает в себя все элементы, которые есть в обоих множествах:

```
intersection = my_set1.intersection(my_set2)
```

Или аналог:

```
intersection = my_set1 & my_set2
```

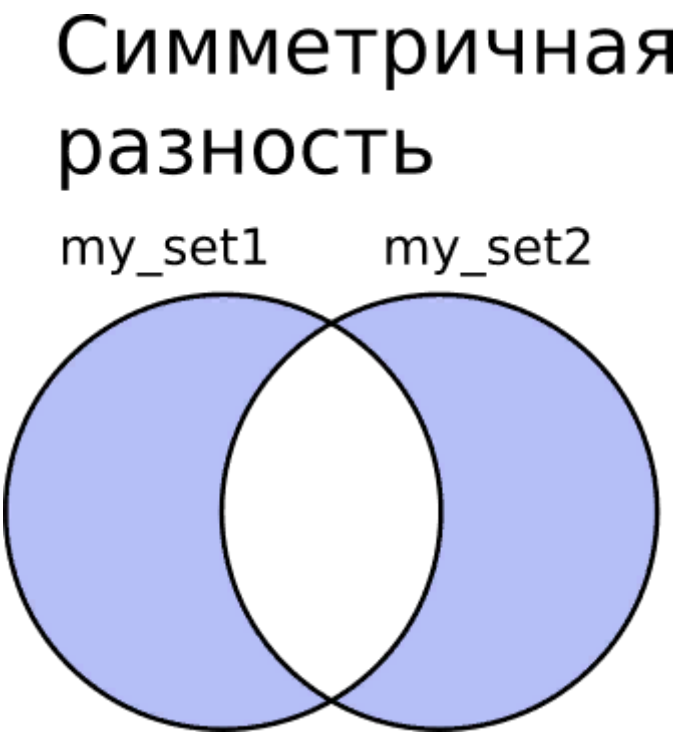


Разность двух множеств включает в себя все элементы, которые есть в первом, и которых нет во втором

```
diff = my_set1.difference(my_set2)
```

Или аналог:

```
diff = my_set1 - my_set2
```



Симметричная разность двух множеств включает в себя все элементы, которые есть только в одном из этих множеств:

```
symm_diff = my_set1.symmetric_difference(my_set2)
```

Или аналогичный вариант:

```
symm_diff = my_set1 ^ my_set2
```

Люди часто путают обозначения | и &, поэтому рекомендуется вместо них писать `s1.union(s2)` и `s1.intersection(s2)`. Операции - и ^ перепутать сложнее, их лучше записывать прямо так.

```
s1 = {'a', 'b', 'c'}
s2 = {'a', 'c', 'd'}
union = s1.union(s2)                # {'a', 'b', 'c', 'd'}
intersection = s1.intersection(s2)  # {'a', 'c'}
diff= s1 - s2                       # {'b'}
symm_diff = s1 ^ s2                 # {'b', 'd'}
```

Сравнение множеств

Все операторы сравнения множеств, а именно ==, <, >, <=, >=, возвращают `True`, если сравнение истинно, и `False` в противном случае.

Множества считаются равными, когда они содержат одинаковые наборы элементов. Равенство множеств, как и ожидалось, обозначается оператором ==:

```
if set1 == set2:
    print('Множества равны')
else:
    print('Множества не равны')
```

Обратите внимание, что у двух равных множеств могут быть разные порядки обхода, например, из-за того, что элементы в каждое из них добавлялись в разном порядке.

Неравенство множеств обозначается оператором `!=`. Он работает противоположно оператору ==.

Теперь перейдём к операторам <=, >=. Они означают «является подмножеством» и «является надмножеством». Подмножество — это некоторая выборка элементов множества, которая может быть как меньше множества, так и совпадать с ним, на что указывают символы “<” и “=” в операторе <=. Наоборот, надмножество включает все элементы некоторого множества и, возможно, какие-то ещё.

```
s1 = {'a', 'b', 'c'}
print(s1 <= s1)  # True

s2 = {'a', 'b'}
print(s2 <= s1)  # True
s3 = {'a'}
print(s3 <= s1)  # True
s4 = {'a', 'z'}
print(s4 <= s1)  # False
```

Операция `s1 < s2` означает «s1 является подмножеством s2, но целиком не совпадает с ним».
Операция `s1 > s2` означает «s1 является надмножеством s2, но целиком не совпадает с ним».

Преобразования между списком и множеством

Функция `set` преобразует всё, что можно обойти с помощью цикла `for`, в множество:

```
set1 = set([1, 2, 'hello'])
set2 = set(range(200))  # множество чисел от 0 до 199 включительно
```

При этом повторяющиеся значения будут входить в множество только один раз.

Множество можно преобразовать в список с помощью функции `list`. Порядок преобразования случайный.

```
my_set = {'a', 'b', 'c'}
lst = list(my_set)
print(lst) # => ['b', 'c', 'a']
```

Нетрудно догадаться, что если нам нужно удалить дубликаты из какого-то списка и при этом необязательно сохранять порядок элементов в нём, это можно сделать так:

```
lst = list(set(lst))
```

Конечно, это сработает только если элементы `lst` неизменяемые.