Элементы теории множеств. Множества в Python

Аннотация

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

1.Теория множеств

Теория множеств — раздел математики, в котором изучаются общие свойства множеств — совокупностей элементов произвольной природы, обладающих каким-либо общим свойством. В основе теории множеств лежат первичные понятия: принадлежность элемента множеству, пустое множество, универсальное множество, подмножество и надмножество.

В программировании используются множества как математические объекты, над которыми можно производить некоторые логические операции.

Мы написали уже много программ, работающих с данными, количество которых неизвестно на момент написания программы. Теперь было бы здорово уметь хранить в памяти неизвестное на момент написания программы количество данных. В этом нам помогут так называемые **коллекции** — специальные типы данных, которые умеют хранить несколько значений под одним именем. Первая из коллекций, с которой мы познакомимся, называется **множество**.

Множество

Множество — составной тип данных, представляющий собой несколько значений (элементов множества) под одним именем. Этот тип называется `set`, не создавайте, пожалуйста, переменные с таким именем! Чтобы задать множество, нужно в фигурных скобках перечислить его элементы.

Здесь создается множество из четырех элементов (названий млекопитающих), которое затем выводится на экран:

```
mammals = {'cat', 'dog', 'fox', 'elephant'}
print(mammals)
```

Введите этот код в IDE и запустите программу несколько раз. Скорее всего, вы увидите разный порядок перечисления млекопитающих, так происходит потому, что элементы в множестве Python не упорядочены. Это позволяет быстро выполнять операции над множествами, о которых мы скоро поговорим чуть позже.

Создание множества

Для создания пустых множеств обязательно вызывать функцию set:

```
empty = set()
```

Обратите внимание: элементами множества могут быть строки или числа. Возникает вопрос: а может ли множество содержать и строки, и числа? Давайте попробуем:

```
mammals_and_numbers = {'cat', 5, 'dog', 3, 'fox', 12, 'elephant', 4}
print(mammals_and_numbers)
```

Как видим, множество может содержать и строки, и числа, а Python опять выводит элементы множества в случайном порядке. Заметьте, если поставить в программе оператор вывода множества на экран несколько раз, не изменяя само множество, порядок вывода элементов не изменится.

Может ли элемент входить в множество несколько раз? Это было бы странно, так как совершенно непонятно, как отличить один элемент от другого. Нет смысла хранить несколько одинаковых объектов, удобно иметь контейнер, сохраняющий только уникальный объекты. Поэтому множество содержит каждый элемент только один раз. Следующий фрагмент кода это демонстрирует:

```
birds = {'raven', 'sparrow', 'sparrow', 'dove', 'hawk', 'falcon'}
print(birds)
```

Важно!

Итак, у множеств есть три ключевые особенности:

Порядок элементов в множестве не определен

Элементы множеств — строки и/или числа

Множество не может содержать одинаковых элементов

Выполнение этих трех свойств позволяет организовать элементы множества в структуру со сложными взаимосвязями, благодаря которым можно быстро проверять наличие элементов в множестве, объединять множества и т. д. Но пока давайте обсудим ограничения.

2.Операции над множеством

Простейшая операция — **вычисление числа элементов** множества. Для этого служит функция `len`. Мы уже встречались с этой функцией раньше, когда определяли длину строки:

```
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 в множестве выбирается случайным образом при запуске интерпретатора Python.

Очень часто необходимо **обойти все элементы** множества в цикле. Для этого используется цикл `for` и оператор `in`, с помощью которых можно перебрать не только все элементы диапазона (как мы это делали раньше, используя `range`), но и элементы множества:

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

такой код выводит:

```
b
a
c
```

Однако, как и в прошлый раз, 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, порядок обхода элементов тоже не изменится

-После изменения множества порядок элементов может измениться произвольным образом

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

```
if elem in my_set:
    print('Элемент есть в множестве')
else:
    print('Элемента нет в множестве')
```

Выражение `elem in my_set` возвращает True, если элемент есть в множестве, и False, если его нет. Интересно, что эта операция для множеств в Python выполняется за время, не зависящее от мощности множества (количества его элементов).

Добавление элемента в множество делается при помощи `add`:

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

`add` — что-то вроде функции, «приклеенной» к конкретному множеству. Такие «приклеенные функции» называются **методами**.

Таким образом, если в коде присутствует имя множества, затем точка и еще одно название со скобками, второе название — имя метода. Если элемент, равный `new_elem`, уже существует в множестве, оно не изменится, поскольку не может содержать одинаковых элементов. Ошибки при этом не произойдет. Небольшой пример:

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

Данный код три раза вызовет метод `add`, «приклеенный» к множеству `my_set`, а затем выведет либо `{'a', 'b'}`, либо `{'b', 'a'}`.

С **удалением элемента** сложнее. Для этого есть сразу три метода:

discard (удалить заданный элемент, если он есть в 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 % уверены, что элемент должен быть в множестве, то лучше получить ошибку во время отладки и исправить ее, 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, произойдет ошибка KeyError.

Очистить множество от всех элементов можно методом clear:

```
my_set.clear()
```

3. 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)
```

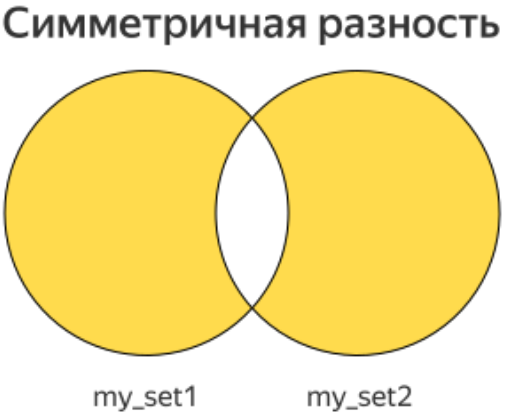
Или аналог:
```
intersection = my_set1 & my_set2
```

Разность двух множеств включает в себя все элементы, которые есть в первом множестве, но 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 можно 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'}
```

4. Сравнение множеств

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

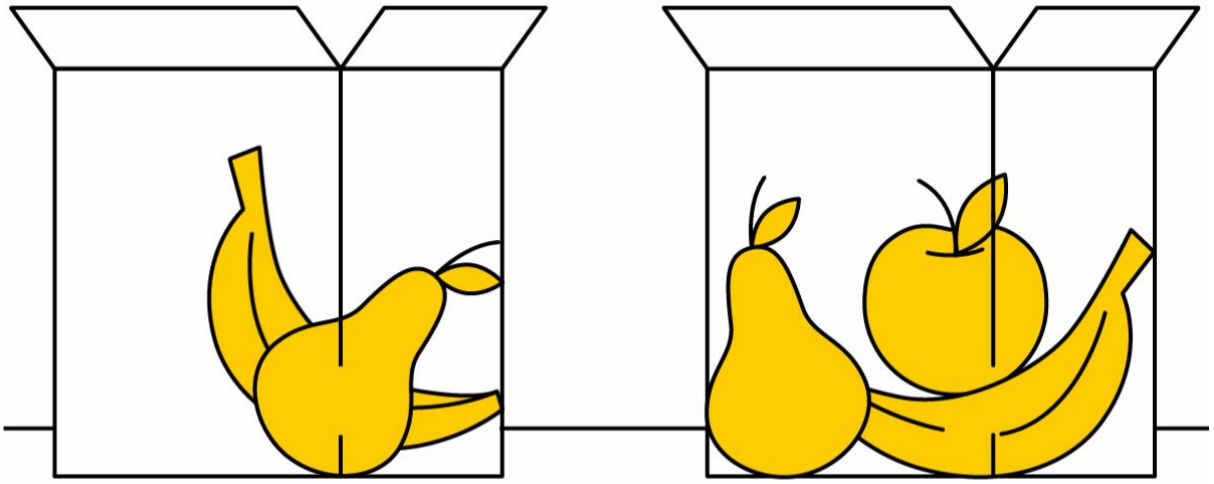
Равенство и неравенство множеств

Множества считаются равными, если они содержат одинаковые наборы элементов. Равенство множеств, как в случае с числами и строками, обозначается оператором ==.

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

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

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



Теперь перейдем к операторам <=, >=. Они означают «является подмножеством» и «является надмножеством».

Подмножество и надмножество

Подмножество — 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 еще.

```
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».