Множества (set)

Мы уже знаем списки и кортежи - упорядоченные структуры, которые могут хранить в себе объекты любых типов, к которым мы можем обратиться по индексу. Теперь поговорим о стуктурах неупорядоченных - множествах и словарях.

Множества хранят некоторое количество объектов, но, в отличие от списка, один объект может храниться в множестве не более одного раза. Кроме того, порядок элементов множества произволен, им нельзя управлять.

Тип называется set, это же является конструктором типа, т.е. в функцию set можно передать произвольную последовательность, и из этой последовательности будет построено множество:

In [1]:
```
x = set()
type(x)
```

Out[1]:
```
'/Users/ianpile/DPO 2020'
```

In [6]:
```
x = (4, 5, 6)
print(set(x)) # передаем список
```

```
{4, 5, 6}
```

In [9]:
```
set(range(10))
```

Out[9]:
```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

In [25]:
```
print(set()) # передаем tuple
print(set(range(10)))
print(set()) # пустое множество
```

```
{10, 20, 30}
{4, 5, 6}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
set()
```

Другой способ создать множество - это перечислить его элементы в фигурных скобках (список - в квадратных, кортеж в круглых, а множество - в фигурных)

In [10]:
```
primes = {2, 3, 5, 7}
animals = {"cat", "dog", "horse", 'cat'}

print(primes)
print(animals)
```

```
{2, 3, 5, 7}
{'horse', 'cat', 'dog'}
```

In [12]:
```
x = [1,2,3,2,2,2,2,2,2,2,'dog',1]
print(set(x))
```

```
{1, 2, 3, 'dog'}
```

Кстати, обратите внимание, что множество может состоять только из уникальных объектов. Выше множество animals включает в себя только одну кошку несмотря на то, что в конструктор мы передали 'cat' два раза. Преобразовать в список в множество - самый простой способ узнать количество уникальных объектов.

Со множествами работает почти всё, что работает с последовательностями (но не работают индексы, потому что элементы не хранятся упорядоченно).

In [13]:
```
print(len(primes))
```

```
4
```

In [14]:
```
primes = {1,11,22,34,5}
11 in primes
```

Out[14]:
```
True
```

In [15]:
```
animals = {"cat", "dog", "horse", 'cat'}
"cow" in animals
```

Out[15]:
```
False
```

In [67]:
```
 # длина
print(11 in primes) # проверка на наличие элемента in хорошо и быстро работает для м
print("cow" in animals)
```

```
4
False
False
```

Все возможные операции с множествами:
https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

Отдельно мы посмотрим на так называемые операции над множествами. Если вы знаете круги Эйлера, то помните как различают объекты множеств - пересечение, объекты, которые принадлежат множеству a, но не принадлежат b и так далее. Давайте посмотрим, как эти операции реализовани в питоне.

In [17]:
```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
c = {2, 3}
```

In [18]:
```
print(c <= a)
```

```
True
```

In [19]:
```
print(c >= a)
```

```
False
```

In [20]:
```
print(a | b)
```

```
{1, 2, 3, 4, 5, 6}
```

In [22]:
```
a.intersection(b)
```

Out[22]:
```
{3, 4}
```

In [23]:
```
a.union(b)
```

Out[23]:
```
{1, 2, 3, 4, 5, 6}
```

In [24]:
```
a^b
```

Out[24]:
```
{1, 2, 5, 6}
```

In [28]:
```
b.difference(a)
```

Out[28]:
```
{5, 6}
```

In [13]:
```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}
c = {2, 3}

 # проверка на подмножество (с подномжество a)
print(c <= b) # не подмножество, т.к. в b нет 2
print(a >= c)
print(a | b) # объединение a.union(b) aka a+b
print(a & b) # пересечение a.intersection(b)
print(a - b) # разность множеств (все что в a, кроме b) a.difference(b)
print(a ^ b) # симметрическая разность множеств (объединение без пересечения)

c = a.copy() # копирование множества, или set(a)
print(c)
```

```
True
False
True
{1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5, 6}
{3, 4}
{1, 2}
{1, 2, 5, 6}
{1, 2, 3, 4}
```

In [29]:
```
a = {1, 2, 3, 4}
c = a.copy()
c
```

Out[29]:
```
{1, 2, 3, 4}
```

In [11]:
```
print(c.issubset(a)) # c <= a
print(c.isdisjoint(a)) # a и c не пересекаются?
print(a.issuperset(c)) # a включает в себя c как подмножество
```

```
True
False
```

```
True
```

In [30]:
```
s = {1,2,3}
s
```

Out[30]:
```
{1, 2, 3}
```

In [31]:
```
s.add(10)
s
```

Out[31]:
```
{1, 2, 3, 10}
```

In [34]:
```
s.add(10)
s
```

Out[34]:
```
{1, 2, 3, 10}
```

In [36]:
```
s.discard(500)
s
```

Out[36]:
```
{1, 2, 3, 10}
```

In [37]:
```
x = s.pop()
print(s)
print(x)
```

```
{1, 2, 3}
10
```

In [38]:
```
s
```

Out[38]:
```
{1, 2, 3}
```

In [39]:
```
s.clear()
s
```

Out[39]:
```
set()
```

Предыдущие операции не меняли множества, создавали новые. А как менять множество:

In [28]:
```
s = {1, 2, 3}
s.add(10) # добавить
print(s) # обратите внимание, что порядок элементов непредсказуем
s.remove(1) # удаление элемента
s.discard(1) # аналогично, но не будет ошибки, если вдруг такого элемента нет в множ
print(s)
x = s.pop() # удаляет и возвращает один произвольный элемент множества (можем сохран
print(s)
print(x)
s.clear() # очистить
print(s)
```

```
{10, 1, 2, 3}
{10, 2, 3}
```

```
{2, 3}
10
set()
```

In [40]:
```
x = 1
x +=1
x
```

Out[40]:
```
2
```

In [41]:
```
s = {1,2}
# s |= {10, 20}
s = s|{10, 20}
s
```

Out[41]:
```
{1, 2, 10, 20}
```

Как мы сокращали арифметические операции раньше (например, +=), так же можно сокращать операции над множествами.

In [70]:
```
s |= {10, 20} # s = s | {10, 20} # объединение множества s с {10,20}
print(s)
# s ^=, s &= и т.п.
```

```
{10, 20}
```

Словари (dict)

Обычный массив (в питоне это список) можно понимать как функцию, которая сопоставляет начальному отрезку натурального ряда какие-то значения.

Давайте посмотрим на списки непривычным способом. Списки - это функции (отображения), которые отображают начальный ряд натуральных чисел в объекты (проще говоря - преводят число 0,1,2,3... во что-то):

In [29]:
```
l = [10, 20, 30, 'a']
print(l[0])
print(l[1])
print(l[2])
print(l[3])
```

```
10
20
30
a
```

В словарях отображать можно не только начала натурального ряда, а произвольные объекты. Представьте себе настоящий словарь или телефонную книжку. Имени человека соответствует номер телефона.

Классическое использование словарей в анализе данных: хранить частоту слова в тексте.

кот $\longrightarrow$ 10

и $\longrightarrow$ 100

Тейлора → 2

Словарь состоит из набора ключей и соответствующих им значений. Значения могут быть любыми объектами (также как и в списке, хранить можно произвольные объекты). А ключи могут быть почти любыми объектами, но только неизменяемыми. В частности числами, строками, кортежами. Список или множество не могут быть ключом.

Одному ключу соответствует ровно одно значение. Но одно и то же значение, в принципе, можно сопоставить разным ключам.

In [43]:
```
a = dict()
type(a)
```

Out[43]:
```
dict
```

In [44]:
```
a['chapter1'] = 'ghfdlksgrjkasgdjkagrdjksargs'
a
```

Out[44]:
```
{'chapter1': 'ghfdlksgrjkasgdjkagrdjksargs'}
```

In [46]:
```
a[1] = 'hrjegrejk'
```

In [49]:
```
x = (3,4,5)
a[x] = 'hrjekwslerw'
```

In [52]:
```
x = (1,2,3,'str')
a[x] = 'fuidaslt'
```

In [53]:
```
a
```

Out[53]:
```
{'chapter1': 'ghfdlksgrjkasgdjkagrdjksargs',
 1: 'hrjegrejk',
 (3, 4, 5): 'hrjekwslerw',
 (1, 2, 3, 'str'): 'fuidaslt'}
```

In [30]:
```
a = dict()
a[(2,3)] = [2,3] # кортеж может быть ключом, потому что он неизменямый
a
```

Out[30]:
```
{(2, 3): [2, 3]}
```

In [31]:
```
b = dict()
b[[2,3]] = [2,3] # а список уже нет, получим ошибку
print(b)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-c056762b57aa> in <module>()
      1 b = dict()
----> 2 b[[2,3]] = [2,3] # а список уже нет, получим ошибку
      3 print(b)
```

```
TypeError: unhashable type: 'list'
```

Создание словаря

В фигурных скобках (как множество), через двоеточие ключ:значение

In [55]:
```
d = dict()
d
```

Out[55]:
```
{}
```

In [56]:
```
d1 = {"кот": 10, "и": 100, "Тейлора": 2}
print(d1)
```

```
{'кот': 10, 'и': 100, 'Тейлора': 2}
```

In [57]:
```
d1["кот"]
```

Out[57]:
```
10
```

Через функцию dict(). Обратите внимание, что тогда ключ-значение задаются не через двоеточие, а через знак присваивания. А строковые ключи пишем без кавычек - по сути мы создаем переменные с такими названиями и присваиваим им значения (а потом функция dict() уже превратит их в строки).

In [58]:
```
d2 = dict(кот=10, и=100, Тейлора=2)
print(d2) # получили тот же результат, что выше
```

```
{'кот': 10, 'и': 100, 'Тейлора': 2}
```

И третий способ - передаем функции dict() список списков или кортежей с парами ключ-значение.

In [60]:
```
d3 = dict([("кот", 10), ("и", 100), ("Тейлора", 2)]) # перечисление (например, списо
print(d3)
```

```
{'кот': 10, 'и': 100, 'Тейлора': 2}
```

Помните, когда мы говорили про списки, мы обсуждали проблему того, что важно создавать именно копию объекта, чтобы сохранять исходный список. Копию словаря можно сделать так

In [61]:
```
d4 = dict(d3) # фактически, копируем dict который строчкой выше
print(d4)
```

```
{'кот': 10, 'и': 100, 'Тейлора': 2}
```

In [62]:
```
d1 == d2 == d3 == d4 # Содержание всех словарей одинаковое
```

Out[62]:
```
True
```

Пустой словарь можно создать двумя способами.

In [63]:
```

```

```
d2 = {} # это пустой словарь (но не пустое множество)
d4 = dict()
print(d2, d4)
```

```
{} {}
```

In [64]:
```
x = {}
type(x)
```

Out[64]:
```
dict
```

Операции со словарями

Как мы уже говорили, словари неупорядоченные структуры и обратиться по индексу к объекту уже больше не удастся.

In [57]:
```
d1[1] # выдаст ошибку во всех случаях кроме того, если в вашем словаре вдруг есть клю
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-57-627b04325301> in <module>()
----> 1 d1[1] # выдаст ошибку во всех случаях кроме того, если в вашем словаре вдруг
 есть ключ 1

KeyError: 1
```

Но можно обращаться к значению по ключу.

In [65]:
```
d3 = dict([("кот", 10), ("и", 100), ("Тейлора", 2)])
print(d1['кот'])
```

```
10
```

In [66]:
```
d1[1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-66-b333f2df0ab4> in <module>
----> 1 d1[1]

KeyError: 1
```

Можно создать новую пару ключ-значение. Для этого просто указываем в квадратных скобках название нового ключа.

In [67]:
```
d1[1] = 'test'
print(d1[1]) # теперь работает!
```

```
test
```

Внимание: если элемент с указанным ключом уже существует, новый с таким же ключом не добавится! Ключ – это уникальный идентификатор элемента. Если мы добавим в словарь новый элемент с уже существующим ключом, мы просто изменим старый – словари являются изменяемыми объектами.

In [68]:
```
d3 = dict([("кот", 10), ("и", 100), ("Тейлора", 2)])
d1["кот"] = 11 # так же как в списке по индексу - можно присвоить новое значение по
```

```
d1
```

Out[68]:
```
{'кот': 11, 'и': 100, 'Тейлора': 2, 1: 'test'}
```

In [70]:
```
d1["кот"] += 1 # или даже изменить его за счет арифметической операции
d1
```

Out[70]:
```
{'кот': 13, 'и': 100, 'Тейлора': 2, 1: 'test'}
```

А вот одинаковые значения в словаре могут быть.

In [71]:
```
d1['собака'] = 13
print(d1)
```

```
{'кот': 13, 'и': 100, 'Тейлора': 2, 1: 'test', 'собака': 13}
```

Кроме обращения по ключу, можно достать значение с помощью метода .get(). Отличие работы метода в том, что если ключа еще нет в словаре, он не генерирует ошибку, а возвращает объект типа None ("ничего"). Это очень полезно в решении некоторых задач.

In [73]:
```
d1
```

Out[73]:
```
{'кот': 13, 'и': 100, 'Тейлора': 2, 1: 'test', 'собака': 13}
```

In [ ]:
```
print(d1['кот'])
```

In [74]:
```
print(d1.get("ктоо")) # вернут None
```

```
None
```

Удобство метода .get() заключается в том, что мы сами можем установить, какое значение будет возвращено, в случае, если пары с выбранным ключом нет в словаре. Так, вместо None мы можем вернуть строку Not found, и ломаться ничего не будет:

In [75]:
```
print(d1.get("ктоо", 'Not found')) # передаем вторым аргументом, что возвращать
print(d1.get("ктоо", False)) # передаем вторым аргументом, что возвращать
```

```
Not found
False
```

Также со словарями работают уже знакомые нам операции - проверка количества элементов, проверка на наличие объектов.

In [ ]:
```
d1
```

In [76]:
```
print(d1)
print("кот" in d1) # проверка на наличие ключа
print("ктоо" not in d1) # проверка на отсутствие ключа
```

```
{'кот': 13, 'и': 100, 'Тейлора': 2, 1: 'test', 'собака': 13}
True
True
```

Удалить отдельный ключ или же очистить весь словарь можно специальными операциями.

In [77]:
```
del d1["кот"]
```

In [78]:
```
d1
```

Out[78]:
```
{'и': 100, 'Тейлора': 2, 1: 'test', 'собака': 13}
```

In [79]:
```
d1.clear()
d1
```

Out[79]:
```
{}
```

In [47]:
```
del d1["кот"] # удалить ключ со своим значением
print(d1)
d1.clear() # удалить все
print(d1)
```

```
{'и': 100, 'Тейлора': 2, 1: 'test'}
{}
```

In [83]:
```
d1 = dict([("кот", 10), ("и", 10), ("Тейлора", 2)])
```

У словарей есть три метода, с помощью которых мы можем сгенерировать список только ключей, только значений и список пар ключ-значения (на самом деле там несколько другая структура, но ведет себя она очень похоже на список).

In [84]:
```
print(d1.values()) # только значения
print(d1.keys()) # только ключи
print(d1.items()) # только ключ-значение
```

```
dict_values([10, 10, 2])
dict_keys(['кот', 'и', 'Тейлора'])
dict_items([('кот', 10), ('и', 10), ('Тейлора', 2)])
```

Ну, и раз уж питоновские словари так похожи на обычные, давайте представим, что у нас есть словарь, где все слова многозначные. Ключом будет слово, а значением – целый список.

In [88]:
```
my_dict = {'swear' : {'swear' : ['клясться', 'ругаться'], 'dream' : ['спать', 'мечта
```

По ключу мы получим значение в виде списка:

In [91]:
```
my_dict['swear']['swear'][0]
```

Out[91]:
```
'клясться'
```

Так как значением является список, можем отдельно обращаться к его элементам:

In [27]:
```
my_dict['swear'][0] # первый элемент
```

Out[27]:
```
'клясться'
```

Можем пойти дальше и создать словарь, где значениями являются словари! Например,

представим, что в некотором сообществе проходят выборы, и каждый участник может проголосовать за любое число кандидатов. Данные сохраняются в виде словаря, где ключами являются имена пользователей, а значениями – пары *кандидат-голос*.

In [28]:
```
votes = {'user1': {'cand1': '+', 'cand2': '-'},
         'user2' : {'cand1': 0, 'cand3' : '+'}} # '+' - за, '-' - против, 0 - воздер
```

In [29]:
```
votes
```

Out[29]:
```
{'user1': {'cand1': '+', 'cand2': '-'}, 'user2': {'cand1': 0, 'cand3': '+'}}
```

По аналогии с вложенными списками по ключам мы сможем обратиться к значению в словаре, который сам является значением в `votes` (да, эту фразу нужно осмыслить):

In [30]:
```
votes['user1']['cand1'] # берем значение, соответствующее ключу user1, в нем – ключу
```

Out[30]:
```
'+'
```

In [1]:
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = dict1.copy()
dict3.update(dict2)
print(dict3)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

In [2]:
```
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
dict3 = {**dict1, **dict2}
print(dict3)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Через занятие мы с вами начнем работать с условными операторами и циклом for, и тогда мы поговорим об особенностях обращения к значениям в цикле и сортировок словарей.