

# Лекция 8

## Продвинутые коллекции

- frozenset – суть, основные методы, пример использования

Модуль collections

- Counter – суть, основные методы, пример использования
- defaultdict – суть, основные методы, несколько примеров использования (можно из документации)
- OrderedDict – суть, основные методы, несколько примеров использования (нужны хорошие!)
- namedtuple() – суть, основные методы, несколько примеров использования (нужны хорошие!); про преимущества и недостатки по

сравнению со словарем; (классом?); кортежем

Модуль enum:

<https://docs.python.org/3/library/enum.html> (<https://docs.python.org/3/library/enum.html>)

<https://pymotw.com/3/enum/> (<https://pymotw.com/3/enum/>)

## Frozen set

Frozen set - "замороженное множество". Отличается от обычного множества set тем, что не может быть изменено после создания, а также является хэшируемым, то есть может служить ключом в словаре или входить в другое множество.

Напоминание: Обычное множество set - неупорядоченный набор различных хэшируемых элементов. Поддерживает математические операции объединения, пересечения, разности и др. Так как множество не упорядочено, то над ним недоступны операции индексирования и получения срезов.

```
In [1]: a = set('qwerty') # создание обычного множества

In [2]: a
Out[2]: {'e', 'q', 'r', 't', 'w', 'y'}

In [3]: b = frozenset('qwerty') # создание "замороженного множества"

In [5]: b
Out[5]: frozenset({'e', 'q', 'r', 't', 'w', 'y'})

In [6]: # сравнение множеств любого типа идет поэлементно:
# если элементы в обоих множествах одинаковые,
# то множества считаются равными (даже если они разных типов set и frozenset)
a == b
Out[6]: True

In [8]: a-b
Out[8]: set()

In [9]: # операции над множествами могут применяться над множествами любых типов
type(a - b) # разность множеств set и frozenset - тип set!
Out[9]: set

In [10]: type(a | b) # объединение множеств set и frozenset - тип set!
Out[10]: set

In [12]: # если операция выполняется между множествами frozenset, то результат - тоже frozenset
type(frozenset(a) & b)
Out[12]: frozenset

In [13]: a.add(1) # в set можно добавлять элементы
a
Out[13]: {'w', 1, 'r', 'q', 'y', 'e', 't'}
```

```
In [14]: b.add(1) # в frozenset добавлять элементы нельзя!

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-14-00401f31fe08> in <module>()
----> 1 b.add(1) # в frozenset добавлять элементы нельзя!

AttributeError: 'frozenset' object has no attribute 'add'
```

## Модуль collections

Модуль **collections** содержит специализированные типы контейнеров данных, которые можно использовать в качестве альтернативы контейнерам общего назначения Python (dict, tuple, list и set). <https://docs.python.org/3.3/library/collections.html> (<https://docs.python.org/3.3/library/collections.html>)

```
In [20]: import collections
```

### collections.Counter([iterable-or-mapping])

Counter ("счетчик") - вид словаря, который предназначен для подсчета количества хэшируемых объектов. Ключами счетчика являются хэшируемые объекты (в частном случае, неизменяемые, такие как примитивные типы данных, кортежи, frozenset), а значениями - их количество. Причем количество может быть любым целым числом (в т.ч. отрицательным).

Работа с Counter аналогична работе с обычным словарем dict, с отличиями:

- При создании очередного счетчика на вход конструктору можно передать итерируемый объект из хэшируемых элементов (например, список из чисел или строку), в случае чего счетчик сам рассчитывает количество разных элементов в этом объекте.
- При попытке получить значение ключа, который отсутствует в счетчике, возвращается 0 (а не KeyError, как для обычных словарей).
- Имеет несколько специфичных функций (методов).

```
In [18]: set(list("ababababfdscabacs"))
Out[18]: {'a', 'b', 'c', 'd', 'f', 's'}

In [21]: c = collections.Counter() # новый пустой счетчик
c
Out[21]: Counter()

In [22]: c = collections.Counter("ababababfdscabacs") # новый счетчик из итерируемого объекта
c
Out[22]: Counter({'a': 6, 'b': 5, 'c': 2, 'd': 1, 'f': 1, 's': 2})

In [29]: c['x']
Out[29]: 0

In [30]: c['a']
Out[30]: 6

In [25]: d1 = dict(c)
d1
Out[25]: {'a': 6, 'b': 5, 'c': 2, 'd': 1, 'f': 1, 's': 2}

In [28]: print(d1.get('x'))
None

In [31]: c = collections.Counter(['aaa', 4, 'bbb', 'bbb', 4, 2, 4]) # новый счетчик из итерируемого объекта
c
Out[31]: Counter({'aaa': 1, 4: 3, 'bbb': 2, 2: 1})

In [32]: c = collections.Counter({'red': 4, 'blue': 2}) # a new counter from a mapping
c
Out[32]: Counter({'blue': 2, 'red': 4})

In [34]: c = collections.Counter(cats=4, dogs='dd') # a new counter from keyword args
c
Out[34]: Counter({'cats': 4, 'dogs': 'dd'})

In [36]: c.update(['cats'])
```

```
In [40]: c['cats'] += 3

In [41]: c

Out[41]: Counter({'cats': 9, 'dogs': 'dd'})

In [16]: c['cows']

Out[16]: 0
```

Методы collections.Counter

```
In [71]: from collections import Counter

In [72]: Counter('aabbcc')

Out[72]: Counter({'a': 2, 'b': 2, 'c': 2})

In [70]: cntr('aaaabb')

Out[70]: Counter({'a': 4, 'b': 2})
```

elements() - возвращает список из элементов счетчика, количество повторений которых соответствует их значениям в счетчике, в произвольном порядке. Если количество в счетчике указано меньше 1, то это значение в результирующий список не выводится.

```
In [42]: c = collections.Counter(a=4, b=3, c=0, d=-2, e=-15, g=3)
list(c.elements())

Out[42]: ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'g', 'g', 'g']

In [43]: for e in c.elements():
print(e)

a
a
a
a
b
b
b
g
g
g
```

most\_common([n]) - Возвращает список из n наиболее часто встречаемых элементов и их количество в порядке уменьшения частоты появления. Если количество одинаковое, то порядок произвольный. Если n не указано, то возвращает все элементы счетчика (тоже в порядке уменьшения количества).

```
In [47]: for e, qty in c.most_common(3):
print(e)

a
b
g
```

subtract([iterable-or-mapping]) - из значений счетчика вычитаются элементы другого итерируемого объекта или сопоставления. Аналогично dict.update(), но вычитает количество вместо замены значений. И входные, и выходные значения могут быть нулевыми или отрицательными.

```
In [50]: c = collections.Counter(a=4, b=2, c=0, d=-2, x=11)
d = collections.Counter(a=1, b=2, c=3, d=4, g=-5, h=0)
c.subtract(d)
c

Out[50]: Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6, 'g': 5, 'h': 0, 'x': 11})

In [54]: c = collections.Counter(a=4, b=2, c=0, d=-2)
d = collections.Counter()
d.subtract(c)
d

Out[54]: Counter({'a': -4, 'b': -2, 'c': 0, 'd': 2})

In [55]: c = collections.Counter(a=4, b=2, c=0, d=-2)
# d = collections.Counter()
c.subtract('aabdddc')
c

Out[55]: Counter({'a': 2, 'b': 1, 'c': -1, 'd': -5})
```

update([iterable-or-mapping]) - функция обычного словаря, которая работает иначе для счетчиков: значения счетчика

складываются со значениями другого итерируемого объекта или сопоставления. Аналогично dict.update(), но суммирует количество вместо замены значений. И входные, и выходные значения могут быть нулевыми или отрицательными.

```
In [56]: c = collections.Counter(a=4, b=2, c=0, d=-2)
d = collections.Counter(a=1, b=2, c=0, d=4, g=-5)
c.update(d)
c
```

Out[56]: Counter({'a': 5, 'b': 4, 'c': 0, 'd': 2, 'g': -5})

```
In [62]: c = collections.Counter(a=4, b=2, c=0, d=-2)
# d = collections.Counter()
c.update(['x', 'x', 'x'])
# c.update('x')
# c.update('x')
c
```

Out[62]: Counter({'a': 4, 'b': 2, 'c': 0, 'd': -2, 'x': 3})

При работе со счетчиками доступны математические операции для их комбинирования для создания "мультимножеств" (счетчиков только с количествами больше 0). Сложение и вычитание счетчиков складывает или вычитает соответствующие значения, пересечение и объединение возвращают минимальные или максимальные количества, соответственно.

```
In [23]: c = collections.Counter(a=4, b=2, c=0, d=-2)
d = collections.Counter(a=1, b=2, c=0, d=4, g=-5)
```

```
In [24]: c + d
```

Out[24]: Counter({'a': 5, 'b': 4, 'd': 2})

```
In [63]: c - d
```

Out[63]: Counter({'a': 3, 'g': 5, 'x': 3})

```
In [64]: c & d
```

Out[64]: Counter({'a': 1, 'b': 2})

```
In [65]: c | d
```

Out[65]: Counter({'a': 4, 'b': 2, 'd': 4, 'x': 3})

```
In [66]: +c # операция, позволяющая вернуть счетчик без учета количеств меньше 1 (осуществляет сложение с пустым словарем)
```

Out[66]: Counter({'a': 4, 'b': 2, 'x': 3})

```
In [67]: -c # операция, позволяющая вернуть счетчик с инвертированными количествами (осуществляет вычитание из пустого словаря)
```

Out[67]: Counter({'d': 2})

**collections.defaultdict([default\_factory, ]...)**

defaultdict - наследуемый класс Python dict, который принимает default\_factory как первичные аргументы. Тип default\_factory — это обычный тип Python, такой как int или list, но вы также можете использовать функцию или лямбду.

Тип данных, который практически в точности повторяет функциональные возможности словарей, за исключением способа обработки обращений к несуществующим ключам.

Когда происходит обращение к несуществующему ключу, вызывается функция, которая передается в аргументе default\_factory. Эта функция должна вернуть значение по умолчанию, которое затем сохраняется как значение указанного ключа.

Остальные аргументы функции defaultdict() в точности те же самые, что передаются встроенной функции dict().

Объекты типа defaultdict удобно использовать в качестве словаря для слежения за данными.

Например, предположим, что необходимо отслеживать позицию каждого слова в строке s. Ниже показано, насколько просто это можно реализовать с помощью объекта defaultdict:

Пример:  
Счетчик слов в тексте

```
In [1]: #Без defaultdict

sentence = "The red for jumped over the fence and ran to the zoo for food"
words = sentence.split(' ')

reg_dict = {}
for ind, word in enumerate(words):
    if word in reg_dict:
        reg_dict[word] += [ind]
    else:
        reg_dict[word] = [ind]
reg_dict
```

Out[1]: {'The': [0],  
'red': [1],  
'for': [2, 12],  
'jumped': [3],  
'over': [4],  
'the': [5, 10],  
'fence': [6],  
'and': [7],  
'ran': [8],  
'to': [9],  
'zoo': [11],  
'food': [13]}

```
In [3]: # генератор значений по умолчанию:
list()
```

Out[3]: []

```
In [4]: #С использованием defaultdict
from collections import defaultdict

sentence = "The red for jumped over the fence and ran to the zoo for food"
words = sentence.split(' ')

d = defaultdict(list)
for ind, word in enumerate(words):
    d[word] += [ind]
d
```

Out[4]: defaultdict(list, {  
    'The': [0],  
    'red': [1],  
    'for': [2, 12],  
    'jumped': [3],  
    'over': [4],  
    'the': [5, 10],  
    'fence': [6],  
    'and': [7],  
    'ran': [8],  
    'to': [9],  
    'zoo': [11],  
    'food': [13]})

```
In [5]: sentence = "The red for jumped over the fence and ran to the zoo for food"
words = sentence.split(' ')

d = defaultdict(list)
for ind, word in enumerate(words):
    d[word].append(ind)
d
```

Out[5]: defaultdict(list, {  
    'The': [0],  
    'red': [1],  
    'for': [2, 12],  
    'jumped': [3],  
    'over': [4],  
    'the': [5, 10],  
    'fence': [6],  
    'and': [7],  
    'ran': [8],  
    'to': [9],  
    'zoo': [11],  
    'food': [13]})

**collections.OrderedDict([items])**

Возвращает экземпляр подкласса dict, поддерживающий обычные методы dict. OrderedDict - это упорядоченный словарь, то есть dict, который помнит порядок, в котором были вставлены ключи. Если новая запись перезаписывает существующую запись, исходная позиция вставки в словарь остается неизменной. При удалении записи и повторном ее добавлении в OrderedDict она переместится в конец словаря. Могут использоваться в коде абсолютно аналогично обычным словарям.

```
In [83]: from collections import OrderedDict
```

```
In [84]: d = OrderedDict()
d
```

Out[84]: OrderedDict()

```
In [85]: d = OrderedDict(x=2, z=3, v=4)
d
```

Out[85]: OrderedDict([('x', 2), ('z', 3), ('v', 4)])

```
In [86]: d1 = {'a': 5, 'x': 3, 'm': 6}
d = OrderedDict(d1)
d
```

Out[86]: OrderedDict([('a', 5), ('x', 3), ('m', 6)])

```
In [91]: for k, v in d1.items():
        print(k, v)

a 5
x 3
m 6
```

```
In [87]: d['x']
```

Out[87]: 3

```
In [88]: d['x'] = 5
d
```

Out[88]: OrderedDict([('a', 5), ('x', 5), ('m', 6)])

```
In [89]: del d['x']
d
```

Out[89]: OrderedDict([('a', 5), ('m', 6)])

```
In [90]: d['x'] = 7
d
```

Out[90]: OrderedDict([('a', 5), ('m', 6), ('x', 7)])

**Методы OrderedDict**

popitem(last=True) - возвращает как результат и удаляет из упорядоченного словаря последний элемент, если last=True, и первый, если last=False.

```
In [92]: print(d.popitem())
d

('x', 7)
```

Out[92]: OrderedDict([('a', 5), ('m', 6)])

```
In [93]: d['f'] = 11
```

```
In [94]: d.popitem()
```

Out[94]: ('f', 11)

```
In [95]: print(d.popitem(last=False))
d

('a', 5)
```

Out[95]: OrderedDict([('m', 6)])

move\_to\_end(key, last=True) - перемещает указанный ключ в конец упорядоченного словаря, если last=True, и в начало, если last=False. Возвращает ошибку KeyError, если указанного ключа в словаре нет.

```
In [96]: d = OrderedDict.fromkeys('abcde', 0)
d
```

Out[96]: OrderedDict([('a', 0), ('b', 0), ('c', 0), ('d', 0), ('e', 0)])

```
In [98]: for k, v in d.items():
        print(k, v)

a 0
b 0
c 0
d 0
e 0
```

```
In [100]: d.move_to_end('b')
          ' '.join(d.keys())
```

Out[100]: 'a c d e b'

```
In [101]: d.move_to_end('d', last=False)
          ' '.join(d.keys())
```

Out[101]: 'd a c e b'

```
In [102]: d.move_to_end('x')
```

```
-----
KeyError                                Traceback (most recent call last)
<ipython-input-102-c4959aecfc62> in <module>()
----> 1 d.move_to_end('x')

KeyError: 'x'
```

**collections.namedtuple(typename, field\_names, verbose=False, rename=False)**

```
In [103]: NAME = 0
          VALUE = 1
```

```
In [104]: c1 = ('Masha', 42)
```

```
In [105]: c1[0]
```

Out[105]: 'Masha'

```
In [106]: c1[NAME]
```

Out[106]: 'Masha'

Возвращает новый класс (подкласс) кортежей: именованный кортеж. Аналогичен обычному кортежу, но позволяет обращаться к элементам по названию поля, а не только по индексу. Таким образом, именованный кортеж наделяет позицию в кортеже дополнительным смыслом и делает код прозрачнее (самодокументируемым). При этом по производительности ничем не отличаются от обычных кортежей (не являются более медленными и тяжелыми).

typename - название нового подкласса кортежей. Теоретически может не совпадать с названием переменной, используемой для создания кортежей этого подкласса (то есть это не одно и то же).

fieldnames - имена полей кортежа в виде строки (где названия полей идут через запятую или пробел) либо в виде списка строк (где каждая строка - название поля). Имена должны быть допустимыми идентификаторами Python. Порядок их следования определяет порядок следования элементов кортежа.

rename - если True, то некорректные имена полей из fieldnames при создании нового класса именованных кортежей заменяются на имена по умолчанию (нижнее подчеркивание + индекс поля, например "\_2").

verbose - устаревший параметр. Если True, то на печать выводится определение нового класса. Вместо этой опции лучше выводить параметр \_source.

```
In [107]: from collections import namedtuple
```

```
In [109]: Point = namedtuple('Point', ['x', 'y'])
          p = Point(11, 22)      # создание нового объекта с указанием позиционного параметра и по ключевому слову
          p
```

Out[109]: Point(x=11, y=22)

```
In [110]: p.x
```

Out[110]: 11

```
In [111]: p.y
```

Out[111]: 22

```
In [112]: p2 = Point(y=7, x=11)
```

```
In [113]: p2
```

Out[113]: Point(x=11, y=7)



```
In [114]: p2.x = 8
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-114-c3a1100e9998> in <module>()
----> 1 p2.x = 8

AttributeError: can't set attribute
```

```
In [115]: p[0] + p[1]
```

Out[115]: 33

```
In [116]: a, b = p # стандартная распаковка кортежа
a, b
```

Out[116]: (11, 22)

```
In [117]: p.x + p.y # обращение к элементам кортежа по имени поля
```

Out[117]: 33

```
In [118]: Point2 = namedtuple('Point2', 'x y z')
d = {'x': 11, 'y': 22, 'z':0}
Point2(**d)
```

Out[118]: Point2(x=11, y=22, z=0)

```
In [119]: s = namedtuple('MyPoint', ['x', 'y']) # название класса в документации может отличаться, но это не приветствуется
s(x=11, y=22)
```

Out[119]: MyPoint(x=11, y=22)

```
In [120]: Point = namedtuple('Point', ['x', '2y', 'z', 'x', 'def'])
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-120-ecd6f291737b> in <module>()
----> 1 Point = namedtuple('Point', ['x', '2y', 'z', 'x', 'def'])

C:\ProgramData\Anaconda3\lib\collections\__init__.py in namedtuple(typename, field_names, verbose, rename, module)
    401         if not name.isidentifier():
    402             raise ValueError('Type names and field names must be valid '
--> 403                               'identifiers: %r' % name)
    404         if _iskeyword(name):
    405             raise ValueError('Type names and field names cannot be a '

ValueError: Type names and field names must be valid identifiers: '2y'
```

```
In [121]: Point = namedtuple('Point', ['x', '2y', 'z', 'x', 'def'], rename=True)
Point._fields
```

Out[121]: ('x', '\_1', 'z', '\_3', '\_4')

```
In [122]: # еще пример
NetworkAddress = namedtuple('NetworkAddress', ['hostname', 'port'])
a = NetworkAddress('www.python.org', 80)
a.hostname
```

Out[122]: 'www.python.org'

```
In [123]: a.port
```

Out[123]: 80

```
In [124]: type(a)==tuple
```

Out[124]: False

```
In [125]: isinstance(a, tuple)
```

Out[125]: True

С помощью namedtuple очень удобно считывать данные, например, из csv-файла, чтобы потом обращаться к элементам строк по имени поля (а не по индексу).



```
In [126]: import csv
with open("employees.csv", "r") as f:
    f_csv = csv.reader(f)
    # создаем новый класс именованного кортежа, названия полей которого отражают поля из входного файла csv:
    EmployeeRecord = namedtuple('EmployeeRecord', next(f_csv))
    for line in f_csv: # все остальные строки из входного файла считываем в виде кортежа EmployeeRecord
        empl = EmployeeRecord._make(line)
        print(empl.name, empl.title)
```

Ivan Manager  
Maria Accountant  
Victor Programmer

Методы

Помимо стандартных методов кортежей, именованным кортежам доступно еще несколько специализированных методов и атрибутов, названия которых начинаются с нижнего подчеркивания, чтобы избежать потенциальных конфликтов с названиями полей данного именованного кортежа.

somenamedtuple.\_make(iterable)

Создает новый экземпляр данного именованного кортежа из существующей последовательности или итерируемого объекта.

```
In [127]: Point = namedtuple('Point', ['x', 'y'])
```

```
In [128]: t = [11, 22]
Point._make(t)
```

Out[128]: Point(x=11, y=22)

somenamedtuple.\_asdict()

Возвращает новый OrderedDict, который сопоставляет имена полей кортежа с их значениями.

```
In [130]: p = Point(x=11, y=22)
d1 = p._asdict()
```

```
In [131]: for k, v in d1.items():
    print(k, v)
```

x 11  
y 22

somenamedtuple.\_replace(\*\*kwargs)

Возвращает новый именованный кортеж, у которого значения указанных в качестве аргумента полей заменены на заданные значения.

```
In [132]: p = Point(x=11, y=22)
p2 = p._replace(x=33)
p2
```

Out[132]: Point(x=33, y=22)

```
In [133]: p is p2
```

Out[133]: False

Метод \_replace можно использовать для кастомизации заданного прототипа (кортежа со значениями по умолчанию):

```
In [145]: Account = namedtuple('Account', 'owner balance transaction_count')
default_account = Account('<owner name>', 0.0, 0)
johns_account = default_account._replace(owner='John')
janes_account = default_account._replace(owner='Jane')
```

somenamedtuple.\_source

Возвращает код определения данного класса именованного кортежа (в виде текстовой строки).

```
In [135]: Point._source

Out[135]: "from builtins import property as _property, tuple as _tuple\nfrom operator import itemgetter as _item\ngetter\nfrom collections import OrderedDict\n\nclass Point(tuple):\n    'Point(x, y)'\n\n    __slots__\n= ()\n\n    _fields = ('x', 'y')\n\n    def __new__(_cls, x, y):\n        'Create new instance of Poin\n    t(x, y)'\n        return _tuple.__new__(_cls, (x, y))\n\n    @classmethod\n    def _make(cls, iterabl\n    e, new=tuple.__new__, len=len):\n        'Make a new Point object from a sequence or iterable'\n    result = new(cls, iterable)\n        if len(result) != 2:\n            raise TypeError('Expected 2 arg\n    uments, got %d' % len(result))\n        return result\n\n    def _replace(_self, **kwds):\n        'Re\n    turn a new Point object replacing specified fields with new values'\n        result = _self._make(map\n    (kwds.pop, ('x', 'y'), _self))\n        if kwds:\n            raise ValueError('Got unexpected field n\n    ames: %r' % list(kwds))\n        return result\n\n    def __repr__(self):\n        'Return a nicely fo\n    rmatted representation string'\n        return self.__class__.__name__ + '(x=%r, y=%r)' % self\n\n    def _asdict(self):\n        'Return a new OrderedDict which maps field names to their values.'\n    return OrderedDict(zip(self._fields, self))\n\n    def __getnewargs__(self):\n        'Return self as\n    a plain tuple.  Used by copy and pickle.'\n        return tuple(self)\n\n    x = _property(_itemgetter\n    (0), doc='Alias for field number 0')\n\n    y = _property(_itemgetter(1), doc='Alias for field number\n    1')\n\n"
```

**somenamedtuple.\_fields**

Возвращает кортеж строк с именами полей именованного кортежа. Полезен для создания новых типов именованных кортежей из уже существующих.

```
In [136]: p._fields

Out[136]: ('x', 'y')
```

```
In [137]: Color = namedtuple('Color', 'red green blue')
Pixel = namedtuple('Pixel', Point._fields + Color._fields)
px1 = Pixel(11, 22, 128, 255, 0)
px1
```

```
Out[137]: Pixel(x=11, y=22, red=128, green=255, blue=0)
```

```
In [141]: px2 = Pixel(x=11, y=22, red=128, green=255, blue=0)
px2
```

```
Out[141]: Pixel(x=11, y=22, red=128, green=255, blue=0)
```

Чтобы получить значения полей, чьи имена сохранены как строки, можно использовать функцию `getattr()`:

```
In [136]: getattr(p, 'x')

11
```

```
In [143]: getattr(px1, 'red')

Out[143]: 128
```

```
In [144]: fields = ['red', 'y', 'x']
for field in fields:
    print(getattr(px2, field))

128
22
11
```

**Сравнение namedtuple с другими типами данных**

Кортежи:

- namedtuple позволяет обращаться не только по индексу, но и по именам полей.
- И tuple, и namedtuple по умолчанию не изменяемы, но в namedtuple можно изменять значения с помощью функции `replace()`.
- При этом namedtuple занимает ровно столько же памяти, сколько и обычный кортеж (вся дополнительная информация хранится в определении класса).

Словари:

- В словарях ключами могут быть только хэшируемые объекты, в именованных кортежах названиями полей - только строки (еще более узко).
- Значениями и в словарях, и в именованных кортежах могут быть любые объекты.
- Названия полей в namedtuple упорядочены (причем в порядке, заданном пользователем), а ключи в словаре - нет.
- Значения в словарях легко изменяемы, а namedtuple по умолчанию считается неизменяемым объектом (большая защищенность данных).
- При создании словарей нужно каждый раз указывать все поля. Если же требуется создать несколько объектов, у которых названия полей одни и те же, то чтобы их каждый раз не писать, можно один раз описать их в новом классе именованного кортежа.
- Именованный кортеж занимает меньше памяти, так как не требует хранения названий полей для каждого экземпляра кортежа (в отличие от словарей).

Классы:

- Именованные кортежи могут заменить определение новых классов, если в них есть только фиксированный набор обычно не изменяемых параметров.

## Модуль enum

Перечисление представляет собой набор символических имен (членов), связанных с уникальными постоянными значениями. В пределах перечисления члены могут сравниваться по идентификатору, и само перечисление может быть повторено.

### Содержание модуля

Этот модуль определяет четыре класса перечислений, которые могут использоваться для определения уникальных наборов имен и значений: Enum , IntEnum , Flag и IntFlag . Он также определяет один декоратор, unique() и один помощник, auto

enum.Enum Базовый класс для создания перечислимых констант.

enum.IntEnum Базовый класс для создания перечислимых констант, которые также являются подклассами int .

enum.IntFlag Базовый класс для создания перечислимых констант, которые можно комбинировать с помощью побитовых операторов, не теряя членства в IntFlag . Члены IntFlag также являются подклассами int .

enum. Flag Базовый класс для создания перечислимых констант, которые можно комбинировать с помощью побитовых операций, не теряя членства в Flag

enum. unique ( ) Декоратор класса Enum, который обеспечивает только одно имя, привязан к какому-либо одному значению.

enum.auto Экземпляры заменяются соответствующим значением для членов Enum.

```
In [2]: from enum import Enum
class Color (Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

Значениями элементов могут быть любые: int , str и т. Д. Если точное значение неважно, вы можете использовать auto экземпляры, и для вас будет выбрано соответствующее значение.

Класс Color - это перечисление (или перечисление )  
Атрибуты Color.RED , Color.GREEN и т. Д. Являются членами перечисления (или перечисляющими членами ) и являются функциональными константами.  
Члены перечисления имеют имена и значения (имя Color.RED равно RED , значение Color.BLUE равно 3 и т. Д.).

Несмотря на то, что мы используем синтаксис class для создания Enums, Enums не являются нормальными классами Python.

```
In [3]: print(Color.RED)

Color.RED
```

```
In [5]: print(repr(Color.RED))
#repr Для многих типов функция возвращает строку, которая при передаче в eval() может произвести объект с тем же з
#что и исходный.
#В других случаях представление является строкой, обрамлённой угловыми скобками (< и >),
#содержащей название типа и некую дополнительную информацию, часто — название объекта и его адрес в памяти.

<Color.RED: 1>
```

```
In [8]: type(Color.RED)
```

Out[8]: <enum 'Color'>

```
In [9]: isinstance(Color.GREEN, Color)
```

Out[9]: True

Перечисления поддерживают итерацию в порядке определения:

```
In [10]: class Shake(Enum):
        VANILLA = 7
        CHOCOLATE = 4
        COOKIES = 9
        MINT = 3
        for shake in Shake:
            print(shake)

Shake.VANILLA
Shake.CHOCOLATE
Shake.COOKIES
Shake.MINT

In [13]: #Члены перечисления хешируются, поэтому их можно использовать в словарях и множествах

apples = {}
apples[Color.RED] = 'red delicious'
apples[Color.GREEN] = 'granny smith'
apples == {Color.RED: 'red delicious', Color.GREEN: 'granny smith'}
```

Out[13]: True

## Программный доступ к элементам перечисления и их атрибутам

Иногда полезно обращаться к членам в перечислениях программно (т.е. ситуации, когда Color.RED не будет выполняться, потому что точный цвет не известен во время записи программы). Enum допускает такой доступ

```
In [19]: Color(1)

Out[19]: <Color.RED: 1>
```

Если вы хотите получить доступ к элементам перечисления по имени , используйте доступ к элементу:

```
In [20]: Color['RED']

Out[20]: <Color.RED: 1>
```

Если у вас есть член перечисления и необходимо его name или value

```
In [22]: member = Color.RED
        member.name, member.value

Out[22]: ('RED', 1)
```

Дублирование двух членов перечисления запрещено, будет возникать ошибка TypeError: Attempted to reuse key:

```
In [14]: class Shape ( Enum ) :
        SQUARE = 2
        SQUARE = 3

-----
TypeError                                Traceback (most recent call last)
<ipython-input-14-3d4ef96612ab> in <module>()
----> 1 class Shape ( Enum ) :
      2     SQUARE = 2
      3     SQUARE = 3

<ipython-input-14-3d4ef96612ab> in Shape()
      1 class Shape ( Enum ) :
      2     SQUARE = 2
----> 3     SQUARE = 3

~\Anaconda3\lib\enum.py in __setitem__(self, key, value)
    90         elif key in self._member_names:
    91             # descriptor overwriting an enum?
--> 92         raise TypeError('Attempted to reuse key: %r' % key)
    93         elif not _is_descriptor(value):
    94             if key in self:

TypeError: Attempted to reuse key: 'SQUARE'
```

Однако двум членам перечисления разрешено иметь одинаковое значение. Учитывая два члена А и В с тем же значением (и А, определенным вначале), В является псевдонимом А. Поиск по значению значения А и В вернет А. Поиск по имени В также вернет А

```
In [15]: class Shape(Enum):
        SQUARE = 2
        DIAMOND = 1
        CIRCLE = 3
        ALIAS_FOR_SQUARE = 2
```

```
In [16]: Shape.SQUARE
```

```
Out[16]: <Shape.SQUARE: 2>
```

```
In [17]: Shape.ALIAS_FOR_SQUARE
```

```
Out[17]: <Shape.SQUARE: 2>
```

```
In [18]: Shape(2)
```

```
Out[18]: <Shape.SQUARE: 2>
```