Лекция 7

Advanced collections and other…

- 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 [2]:
```
s1 = set('qwerty') # создание обычного множества
s1
```

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

In [3]:
```
fs1 = frozenset('qwerty') # создание "замороженного множества"
fs1
```

Out[3]:
```
frozenset({'e', 'q', 'r', 't', 'w', 'y'})
```

In [4]:
```
# сравнение множеств любого типа идет поэлементно:
# если элементы в обоих множествах одинаковые,
# то множества считаются равными (даже если они разных типов set и frozenset)
s1 == fs1
```

Out[4]:
```
True
```

In [5]:
```
fs2 = frozenset('qwertyasdf')
fs3 = frozenset('qwer')
fs2, fs3
```

Out[5]:
```
(frozenset({'a', 'd', 'e', 'f', 'q', 'r', 's', 't', 'w', 'y'}),
 frozenset({'e', 'q', 'r', 'w'}))
```

In [6]:
```
# операции над множествами могут применяться над множествами любых типов
dif1 = fs2 - s1
dif1, type(dif1)
```

Out[6]:
```
(frozenset({'a', 'd', 'f', 's'}), frozenset)
```

In [7]:
```
dif1 is s1 # dif1 - это новое frozenset!
```

Out[7]:
```
False
```

In [9]:
```
dif2 = s1 - fs3
dif2, type(dif2)
```

Out[9]:
```
({'t', 'y'}, set)
```

In [10]:
```
s1 is dif2
```

Out[10]:
```
False
```

In [11]:
```
# объединение множеств set и frozenset
s1, fs2, s1 | fs2, fs2 | s1, fs2 | s1 is fs2
```

Out[11]:
```
({'e', 'q', 'r', 't', 'w', 'y'},
 frozenset({'a', 'd', 'e', 'f', 'q', 'r', 's', 't', 'w', 'y'}),
 {'a', 'd', 'e', 'f', 'q', 'r', 's', 't', 'w', 'y'},
 frozenset({'a', 'd', 'e', 'f', 'q', 'r', 's', 't', 'w', 'y'}),
 False)
```

In [12]:
```
# если операция выполняется между множествами frozenset, то результат - тоже frozenset
intersec1 = frozenset(s1) & fs2
intersec1, type(intersec1)
```

Out[12]:
```
(frozenset({'e', 'q', 'r', 't', 'w', 'y'}), frozenset)
```

In [13]:
```
s1.add(1) # в set можно добавлять элементы
s1
```

Out[13]:
```
{1, 'y', 't', 'q', 'r', 'e', 'w'}
```

In [14]:
```
fs1.add(1) # в frozenset добавлять элементы нельзя!
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-14-3122c98c73ee> in <module>()
----> 1 fs1.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/3.6/library/collections.html (https://docs.python.org/3.6/library/collections.html)

In [15]:
```
import collections
```

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

Counter ("счетчик") - вид словаря, который позволяет 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 [16]:
```
c = collections.Counter() # новый пустой счетчик
c
```

Out[16]:
```
Counter()
```

In [17]:
```
c = collections.Counter("abababababfdscabacs") # новый счетчик из итерируемого объекта
c
```

Out[17]:
```
Counter({'a': 6, 'b': 5, 'c': 2, 'd': 1, 'f': 1, 's': 2})
```

In [18]:
```
c = collections.Counter(['aaa', 4, 'bbb', 'bbb', 4, 2, 4]) # новый счетчик из итерируемого объекта
c
```

Out[18]:
```
Counter({'aaa': 1, 4: 3, 'bbb': 2, 2: 1})
```

In [19]:
```
c = collections.Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
c
```

Out[19]:
```
Counter({'blue': 2, 'red': 4})
```

In [20]:
```
c = collections.Counter(cats=4, dogs=8)             # a new counter from keyword args
c
```

Out[20]:
```
Counter({'cats': 4, 'dogs': 8})
```

In [205]:
```
c['cats']
```

Out[205]:
```
4
```

In [21]:
```
c['cows']
```

Out[21]:
```
0
```

Методы collections.Counter

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 [23]:
```
c = collections.Counter(a=4, b=3, c=0, d=-2, e=-15, g=3)
list(c.elements())
```

Out[23]:
```
['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 [28]:
```
c.most_common(2)
```

Out[28]:
```
[('a', 4), ('b', 3)]
```

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 [29]:
```
c = collections.Counter(a=4, b=2, c=0, d=-2)
d = collections.Counter(a=1, b=2, c=3, d=4, g=-5, h=0)
c.subtract(d)
c
```

Out[29]:
```
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6, 'g': 5, 'h': 0})
```

In [32]:
```
c = collections.Counter(a=4, b=2, c=0, d=-2)
d = collections.Counter()
d.subtract(c)
d
```

Out[32]:
```
Counter({'a': -4, 'b': -2, 'c': 0, 'd': 2})
```

update([iterable-or-mapping]) - функция обычного словаря, которая работает иначе для счетчиков: значения счетчика складываются со значениями другого итерируемомо объекта или сопоставления. Аналогично dict.update(), но суммирует количество вместо замены значений. И входные, и выходные значения могут быть нулевыми или отрицательными.

In [33]:
```
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[33]:
```
Counter({'a': 5, 'b': 4, 'c': 0, 'd': 2, 'g': -5})
```

In [46]:
```
c = collections.Counter(a=4, b=2, c=0, d=-2)
d = collections.Counter()
c.update(d)
c
```

Out[46]:
```
Counter({'a': 4, 'b': 2, 'c': 0, 'd': -2})
```

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 [45]:
```
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 [36]:
```
c + d
```

Out[36]:
```
Counter({'a': 5, 'b': 4, 'd': 2})
```

In [37]:
```
c - d
```

Out[37]:
```
Counter({'a': 3, 'g': 5})
```

In [38]:
```
c & d
```

Out[38]:
```
Counter({'a': 1, 'b': 2})
```

In [43]:
```
c | d
```

Out[43]:
```
Counter({'a': 4, 'b': 2, 'd': 4})
```

In [44]:
```
+c # операция, позволяющая вернуть счетчик без учета количеств меньше 1 (осуществляет сложение с пустым словарем)
```

Out[44]:
```
Counter({'a': 4, 'b': 2})
```

In [219]:
```
-c # операция, позволяющая вернуть счетчик с инвертированными количествами (осуществляет вычитание из пустого словаря)
```

Out[219]:
```
Counter({'d': 2})
```

```
In [51]: e = collections.Counter(a=4, b=2, c=0, d=-2)
         e
Out[51]: Counter({'a': 4, 'b': 2, 'c': 0, 'd': -2})

In [52]: e.update('a')
         e
Out[52]: Counter({'a': 5, 'b': 2, 'c': 0, 'd': -2})

In [53]: e['a']=1
         e
Out[53]: Counter({'a': 1, 'b': 2, 'c': 0, 'd': -2})
```

collections.defaultdict([default_factory,]...)

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

Является подклассом обычного словаря dict, который принимает default_factory в качестве первого по порядку аргумента.

default_factory — функция, которая используется при попытке получить из словаря отсутствующее значение. Если ключ отсутствует в defaultdict, то при вызове его значения вызывается функция default_factory без аргументов и данная запись добавляется в словарь (в виде "отсутствующий ключ: default_factory").

default_factory может быть функцией создания стандартных типов Python (например, int или list, которые при вызове без аргументов возвращают 0 или пустой список, соответственно) или любой другой функцией, в т.ч. лямбда-функцией (которую можно вызвать без параметров с возвращением какого-то значения).

Таким образом, когда происходит обращение к несуществующему ключу, вызывается функция, которая передается в аргументе default_factory при определении нового defaultdict. Эта функция должна вернуть значение по умолчанию (т.е. возвращаемое при вызове функции без параметров), которое затем сохраняется как значение указанного ключа.

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

```
In [55]: from collections import defaultdict

In [56]: d = defaultdict(list, x=4, y=7, z=2)
         d
Out[56]: defaultdict(list, {'x': 4, 'y': 7, 'z': 2})

In [57]: d['one']
Out[57]: []

In [58]: d
Out[58]: defaultdict(list, {'one': [], 'x': 4, 'y': 7, 'z': 2})

In [60]: d = defaultdict(lambda: 4, {'x': 4, 'y': 7, 'z': 2})
         d['one']
Out[60]: 4

In [61]: d
Out[61]: defaultdict(<function __main__.<lambda>>, {'one': 4, 'x': 4, 'y': 7, 'z': 2})
```

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 [63]:
```
sentence = "The red for jumped over the fence and ran to the zoo for food"
words = sentence.split(' ')

collections.Counter(words)
```

Out[63]:
```
Counter({'The': 1,
         'and': 1,
         'fence': 1,
         'food': 1,
         'for': 2,
         'jumped': 1,
         'over': 1,
         'ran': 1,
         'red': 1,
         'the': 2,
         'to': 1,
         'zoo': 1})
```

In [64]:
```
# Без defaultdict

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

reg_dict = {}
for word in words:
    if word in reg_dict:
        reg_dict[word] += 1
    else:
        reg_dict[word] = 1
reg_dict
```

Out[64]:
```
{'The': 1,
 'and': 1,
 'fence': 1,
 'food': 1,
 'for': 2,
 'jumped': 1,
 'over': 1,
 'ran': 1,
 'red': 1,
 'the': 2,
 'to': 1,
 'zoo': 1}
```

In [65]:
```
# Без defaultdict

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

reg_dict = {}
for word in words:
        reg_dict[word] = reg_dict.get(word, 0) + 1
reg_dict
```

Out[65]:
```
{'The': 1,
 'and': 1,
 'fence': 1,
 'food': 1,
 'for': 2,
 'jumped': 1,
 'over': 1,
 'ran': 1,
 'red': 1,
 'the': 2,
 'to': 1,
 'zoo': 1}
```

In [66]:
```
#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(int)
for word in words:
    d[word] += 1
d
```

Out[66]:
```
defaultdict(int,
            {'The': 1,
             'and': 1,
             'fence': 1,
             'food': 1,
             'for': 2,
             'jumped': 1,
             'over': 1,
             'ran': 1,
             'red': 1,
             'the': 2,
             'to': 1,
             'zoo': 1})
```

In [68]:
```
#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 i, word in enumerate(words):
    d[word].append(i)
d
```

Out[68]:
```
defaultdict(list,
            {'The': [0],
             'and': [7],
             'fence': [6],
             'food': [13],
             'for': [2, 12],
             'jumped': [3],
             'over': [4],
             'ran': [8],
             'red': [1],
             'the': [5, 10],
             'to': [9],
             'zoo': [11]})
```

Пример: перевод списка кортежей в словарь, значения в котором - список значений из разных кортежей с одинаковым первым элементом, а ключи - уникальные первые элементы кортежей.

In [69]:
```
s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
d = defaultdict(list)
for k, v in s:
    d[k].append(v)
sorted(d.items())
```

Out[69]:
```
[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]
```

Использовать для таких случаев defaultdict быстрее, чем вызывать функцию d.setdefault при каждом добавлении в словарь.

In [70]:
```
d = {}
for k, v in s:
     d.setdefault(k, []).append(v)
list(d.items())
```

Out[70]:
```
[('yellow', [1, 3]), ('blue', [2, 4]), ('red', [1])]
```

То же самое можно реализовать для создания словаря множеств, а не списков:

In [229]:
```
s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
d = defaultdict(set)
for k, v in s:
    d[k].add(v)
sorted(d.items())
```

Out[229]:
```
[('blue', {2, 4}), ('red', {1}), ('yellow', {1, 3})]
```

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 [71]:
```
from collections import OrderedDict
```

In [75]:
```
d = OrderedDict()
d['x']=2
d['z']=3
d['a']=11
d['v']=4
d
```

Out[75]:
```
OrderedDict([('x', 2), ('z', 3), ('a', 11), ('v', 4)])
```

In [76]:
```
for k in d:
    print(k)
```

```
x
z
a
v
```

In [77]:
```
d['x']=0
```

In [78]:
```
for k in d:
    print(k)
```

```
x
z
a
v
```

In [79]:
```
d['f']=17
```

In [80]:
```
for k in d:
    print(k)
```

```
x
z
a
v
f
```

In [83]:
```
del d['x']
d['x']=1
```

In [84]:
```
for k in d:
    print(k)
```

```
z
a
v
f
x
```

In [73]:
```
d = OrderedDict(x=2, z=3, v=4)
d
```

Out[73]:
```
OrderedDict([('x', 2), ('z', 3), ('v', 4)])
```

In [233]:
```
d1 = {'a': 5, 'x': 3, 'm': 6}
d = OrderedDict(d1)
d
```

Out[233]:
```
OrderedDict([('a', 5), ('x', 3), ('m', 6)])
```

In [234]:
```
d['x']
```

Out[234]:
```
3
```

In [235]:
```
d['x'] = 5
d
```

Out[235]:
```
OrderedDict([('a', 5), ('x', 5), ('m', 6)])
```

In [236]:
```
del d['x']
d
```

Out[236]:
```
OrderedDict([('a', 5), ('m', 6)])
```

In [237]:
```
d['x'] = 7
d
```

Out[237]:
```
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 [86]:
```
print(d.popitem())
d
```

```
('f', 17)
```

Out[86]:
```
OrderedDict([('z', 3), ('a', 11), ('v', 4)])
```

In [87]:
```
print(d.popitem(last=False))
d
```

```
('z', 3)
```

Out[87]:
```
OrderedDict([('a', 11), ('v', 4)])
```

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 [88]:
```
d = OrderedDict.fromkeys('abcde', 0)
d
```

Out[88]:
```
OrderedDict([('a', 0), ('b', 0), ('c', 0), ('d', 0), ('e', 0)])
```

In [89]:
```
d.move_to_end('b')
' '.join(d.keys())
```

Out[89]:
```
'a c d e b'
```

In [90]:
```
d.move_to_end('d', last=False)
' '.join(d.keys())
```

Out[90]:
```
'd a c e b'
```

In [91]:
```
d.move_to_end('x')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-91-c4959aecfc62> in <module>()
----> 1 d.move_to_end('x')

KeyError: 'x'
```

collections.namedtuple(typename, field_names, verbose=False, rename=False)

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. Теоретичесски 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 [92]:
```
from collections import namedtuple
```

In [93]:
```
Point = namedtuple('Point', ['x', 'y'])
p = Point(11, 22)      # создание нового объекта с указанием позиционного параметра и по ключевому слову
p
```

Out[93]:
```
Point(x=11, y=22)
```

In [95]:
```
p2 = Point(y=11, x=22)
p2
```

Out[95]:
```
Point(x=22, y=11)
```

In [96]:
```
p[0] + p[1]
```

Out[96]:
```
33
```

In [97]:
```
a, b = p # стандартная распаковка кортежа
a, b
```

Out[97]: (11, 22)

In [98]:
```
p.x + p.y # обращение к элементам кортежа по имени поля
```

Out[98]: 33

In [99]:
```
Point = namedtuple('Point', 'x y')
d = {'x': 11, 'y': 22}
Point(**d)
```

Out[99]: Point(x=11, y=22)

In [101]:
```
Point = namedtuple('MyPoint', ['x', 'y']) # название класса в документации может отличаться, но это не приветст
Point(x=11, y=22)
```

Out[101]: MyPoint(x=11, y=22)

In [102]:
```
Point = namedtuple('Point', ['x', '2y', 'z', 'x', 'def'])
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-102-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, ren
ame, 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 [103]:
```
Point = namedtuple('Point', ['x', '2y', 'z', 'x', 'def'], rename=True)
Point._fields
```

Out[103]: ('x', '_1', 'z', '_3', '_4')

In [111]:
```
# еще пример
NetworkAddress = namedtuple('NetworkAddress', ['hostname', 'port'])
a = NetworkAddress('www.python.org', 80)
a.hostname
```

Out[111]: 'www.python.org'

In [112]:
```
a.port
```

Out[112]: 80

In [113]:
```
a.hostname
```

Out[113]: 'www.python.org'

In [114]:
```
type(a)
```

Out[114]: __main__.NetworkAddress

In [115]:
```
type(a) == tuple
```

Out[115]: False

In [257]:
```
isinstance(a, tuple)
```

Out[257]: 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 [119]:
```
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))
    empl_l =[]
    for line in f_csv: # все остальные строки из входного файла считываем в виде кортежа EmployeeRecord
        empl = EmployeeRecord._make(line)
        empl_l.append(empl)
        print(empl.name, empl.title)
#         break
```

```
Ivan Manager
```

In [120]:
```
line
```

Out[120]:
```
['Ivan', '25', 'Manager', 'Risks', '$1000']
```

In [118]:
```
empl_l[0]
```

Out[118]:
```
EmployeeRecord(name='Ivan', age='25', title='Manager', department='Risks', paygrade='$1000')
```

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 [121]:
```
Point = namedtuple('Point', ['x', 'y'])
```

In [123]:
```
t = [11, 22]
p1 = Point._make(t)
p1
```

Out[123]:
```
Point(x=11, y=22)
```

In [124]:
```
p1.y
```

Out[124]:
```
22
```

somenamedtuple._asdict()

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

In [125]:
```
p = Point(x=11, y=22)
p._asdict()
```

Out[125]:
```
OrderedDict([('x', 11), ('y', 22)])
```

somenamedtuple._replace(kwargs)**

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

In [126]:
```
p = Point(x=11, y=22)
p._replace(x=33)
```

Out[126]:
```
Point(x=33, y=22)
```

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

In [127]:
```
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 [129]:
```
print(Point._source)
```

```
from builtins import property as _property, tuple as _tuple
from operator import itemgetter as _itemgetter
from collections import OrderedDict

class Point(tuple):
    'Point(x, y)'

    __slots__ = ()

    _fields = ('x', 'y')

    def __new__(_cls, x, y):
        'Create new instance of Point(x, y)'
        return _tuple.__new__(_cls, (x, y))

    @classmethod
    def _make(cls, iterable, new=tuple.__new__, len=len):
        'Make a new Point object from a sequence or iterable'
        result = new(cls, iterable)
        if len(result) != 2:
            raise TypeError('Expected 2 arguments, got %d' % len(result))
        return result

    def _replace(_self, **kwds):
        'Return a new Point object replacing specified fields with new values'
        result = _self._make(map(kwds.pop, ('x', 'y'), _self))
        if kwds:
            raise ValueError('Got unexpected field names: %r' % list(kwds))
        return result

    def __repr__(self):
        'Return a nicely formatted representation string'
        return self.__class__.__name__ + '(x=%r, y=%r)' % self

    def _asdict(self):
        'Return a new OrderedDict which maps field names to their values.'
        return OrderedDict(zip(self._fields, self))

    def __getnewargs__(self):
        'Return self as a plain tuple.  Used by copy and pickle.'
        return tuple(self)

    x = _property(_itemgetter(0), doc='Alias for field number 0')

    y = _property(_itemgetter(1), doc='Alias for field number 1')
```

somenamedtuple._fields

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

In [130]:
```
p._fields
```

Out[130]:
```
('x', 'y')
```

In [265]:
```
Color = namedtuple('Color', 'red green blue')
Pixel = namedtuple('Pixel', Point._fields + Color._fields)
px1 = Pixel(11, 22, 128, 255, 0)
px1
```

Out[265]:
```
Pixel(x=11, y=22, red=128, green=255, blue=0)
```

In [266]:
```
px2 = Pixel(x=11, y=22, red=128, green=255, blue=0)
px2
```

Out[266]:
```
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 [267]:
```
getattr(p, 'x')
```

Out[267]:
```
11
```

In [268]:
```
getattr(px1, 'red')
```

Out[268]:
```
128
```

In [269]:
```
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 (enumeration) - это конструкция, позволяющая хранить упорядоченный набор имен (членов перечисления), которым соответствуют уникальные постоянные значения. В пределах отдельного перечисления его члены можно сравнивать на эквивалентность, а также по членам перечисления можно итерироваться (проходить их по порядку).

enum.Enum - базовый класс для создания перечислений. Есть также три дополнительных класса: enum.IntEnum, enum.IntFlag, enum.Flag, и другие возможности. Мы рассмотрим только базовый класс enum.Enum.

Пример перечисления:

In [286]:
```
from enum import Enum
```

In [287]:
```
class Color(Enum):
    RED = 1
    GREEN = 5
    BLUE = 3
Color
```

Out[287]:
```
<enum 'Color'>
```

In [288]:
```
Color.RED
```

Out[288]:
```
<Color.RED: 1>
```

In [289]:
```
Color(5)
```

Out[289]:
```
<Color.GREEN: 5>
```

In [290]:
```
Color.BLUE.name
```

Out[290]:
```
'BLUE'
```

Типы элементов перечисления могут быть любые: int , str и т. д. Если точное значение элемента перечисления не важно (например, важен только состав и порядок, а значения - любые уникальные идентификаторы), то вы можете укзаать enum.auto() в качестве значения.

In [293]:
```
from enum import auto
```

In [323]:
```
class Color(Enum):
    RED = auto()
    GREEN = auto()
    BLUE = auto()
Color.GREEN
```

Out[323]:
```
<Color.GREEN: 2>
```

Терминология для рассмотренного случая:

- Color - это перечисление (enumeration или enum).
- Color.RED, Color.GREEN и т.д - члены перечисления и функционально являются константами.
- Члены перечисления имеют имена и значения (имя Color.RED равно RED, значение Color.BLUE равно 3 и т.д.).

In [296]:
```
print(Color.RED) # человекочитаемое название костанты (члена перечисления)
```

```
Color.RED
```

In [273]:
```
print(repr(Color.RED)) # вывод на печать более подробной информации о члене перечисления
# repr - функция, которая для многих типов данных возвращает строку, которая при передаче в eval()
# может произвести объект с тем же значением, что и у переданного как параметр.
# В других случаях представление repr 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 [298]:
```
repr(x for x in range(3)) # пример repr для генератора: возвращает название типа и адрес
```

Out[298]:
```
'<generator object <genexpr> at 0x0000024B4B374DB0>'
```

In [274]:
```
type(Color.RED)
```

Out[274]:
```
<enum 'Color'>
```

In [275]:
```
isinstance(Color.GREEN, Color)
```

Out[275]:
```
True
```

Перечисления поддерживают итерацию в порядке определения:

In [276]:
```
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 [303]:
```
for shake in sorted(Shake, key=lambda x: x.value):
    print("{} - {}".format(shake.name, shake.value))
```

```
MINT - 3
CHOCOLATE - 4
VANILLA - 7
COOKIES - 9
```

In [304]:
```
#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[304]:
```
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 [308]:
```
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3
```

In [309]:
```
Color(1)
```

Out[309]:
```
<Color.RED: 1>
```

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

In [310]:
```
Color['RED']
```

Out[310]:
```
<Color.RED: 1>
```

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

In [311]:
```
member = Color.RED
member.name, member.value
```

Out[311]:
```
('RED', 1)
```

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

In [312]:
```
class Shape ( Enum ):
    SQUARE = 2
    SQUARE = 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-312-3d4ef96612ab> in <module>()
----> 1 class Shape ( Enum ):
      2     SQUARE = 2
      3     SQUARE = 3

<ipython-input-312-3d4ef96612ab> in Shape()
      1 class Shape ( Enum ):
      2     SQUARE = 2
----> 3     SQUARE = 3

C:\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. Если есть два члена перечисления с одинаковыми значениями A и B, причем A объявлено раньше, то B является псевдонимом A. Поиск по значению значения A и B вернет A. Поиск по имени B также вернет A.

In [317]:
```
class Shape(Enum):
    SQUARE = 2
    DIAMOND = 1
    CIRCLE = 3
    ALIAS_FOR_SQUARE = 2
    ALIAS_FOR_SQUARE_2 = 2
```

In [318]:
```
Shape.SQUARE
```

Out[318]:
```
<Shape.SQUARE: 2>
```

In [319]:
```
Shape.ALIAS_FOR_SQUARE
```

Out[319]:
```
<Shape.SQUARE: 2>
```

In [320]:
```
Shape(2)
```

Out[320]:
```
<Shape.SQUARE: 2>
```

При итерации по перечислению псевдонимы затронуты не будут:

In [324]:
```
list(Shape)
```

Out[324]:
```
[<Shape.SQUARE: 2>, <Shape.DIAMOND: 1>, <Shape.CIRCLE: 3>]
```

Чтобы вывести все члены перечисления, включая псевдонимы, можно воспользоваться атрибутом **members**, который возвращает словарь вида "имя-член_перечисления":

In [328]:
```
Shape.__members__
```

Out[328]:
```
mappingproxy({'ALIAS_FOR_SQUARE': <Shape.SQUARE: 2>,
              'ALIAS_FOR_SQUARE_2': <Shape.SQUARE: 2>,
              'CIRCLE': <Shape.CIRCLE: 3>,
              'DIAMOND': <Shape.DIAMOND: 1>,
              'SQUARE': <Shape.SQUARE: 2>})
```

In [327]:
```
for name, member in Shape.__members__.items():
    print(name, " - ", member)
```

```
SQUARE  -  Shape.SQUARE
DIAMOND  -  Shape.DIAMOND
CIRCLE  -  Shape.CIRCLE
ALIAS_FOR_SQUARE  -  Shape.SQUARE
ALIAS_FOR_SQUARE_2  -  Shape.SQUARE
```

In [329]:
```
# вывести список псевдонимов, используемых в перечислении
[name for name, member in Shape.__members__.items() if member.name != name]
```

Out[329]:
```
['ALIAS_FOR_SQUARE', 'ALIAS_FOR_SQUARE_2']
```

При сравнении членов перечисления следует пользоваться операторами is и is not, то есть сравнение проводиться не на равенство значений, а на идентичность объектов.

In [330]:
```
Color.RED is Color.RED
```

Out[330]:
```
True
```

In [331]:
```
Color.RED is Color.BLUE
```

Out[331]:
```
False
```

In [332]:
```
Color.RED is not Color.BLUE
```

Out[332]:
```
True
```

Хотя операторы сравнения на равенство значений тоже работают. А вот сравнения "больше-меньше" - нет.

In [333]:
```
Color.RED < Color.BLUE
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-333-8fdf89de291e> in <module>()
----> 1 Color.RED < Color.BLUE

TypeError: '<' not supported between instances of 'Color' and 'Color'
```

В классе Enum сравнение на равенство членов перечислений с целыми числами всегда будут выдавать False:

In [334]:
```
Color.BLUE == 2
```

Out[334]:
```
False
```

Значениями членов перечисления могут быть не только числа (хотя часто их значения не нужны для решения задач, и для простоты и удобства используют целые числа):

In [335]:
```
class Fruits ( Enum ):
    FAVOURITE = 'apple'
    GOOD = 'banana'
    EXOTIC = 'mango'
```

In [336]:
```
Fruits.EXOTIC
```

Out[336]:
```
<Fruits.EXOTIC: 'mango'>
```

In [340]:
```
Fruits('banana')
```

Out[340]:
```
<Fruits.GOOD: 'banana'>
```

Более подробно про модуль Enum: https://docs.python.org/3/library/enum.html (https://docs.python.org/3/library/enum.html)