**Модуль collections**

Вы уже хорошо знакомы со встроенными контейнерами данных, такими, как словари, списки, множества и кортежи. Теперь самое время познакомиться со специализированными их альтернативами, которые предоставляются модулем collections.

Вы спросите, зачем нужны специализированные, если есть стандартные? Они расширяют возможности стандартных. Т.е. они как стандартные (полностью) повторяют их возможности, но и добавляют еще некоторые важные дополнительные возможности.

Какая польза от их использования? Оптимизация кода! Вы можете решить задачу и без специализированных коллекций, но объем кода будет больше, чем если бы вы воспользовались collections.

**Counter (Счетчик)**

Например, у вас есть задача выполнить подсчет элементов некой последовательности, например строки, списка и т.д. В обычном случае вам придется написать свое решение и скорее всего вы будете опираться на обычные коллекции. Такое решение будет занимать некоторый объем кода. Но вы можете решить эту задачу буквально в одну строку, воспользовавшись возможностями конструктора Counter.

Он принимает итерируемый объект и возвращает словарь, где ключами являются элементы объекта, а значениями – частоты повторений элемента.

**Листинг 1. task\_1.py**

|  |
| --- |
| *"""Класс collections.Counter()"""* **from** collections **import** Counter  *# создаем объект коллекции* OBJ = Counter([**'js'**, **'java'**, **'java'**, **'python'**, **'python'**, **'python'**]) print(OBJ) *# -> Counter({'python': 3, 'java': 2, 'js': 1})  # объект на базе словаря* print(OBJ[**'python'**]) print(OBJ[**'perl'**])  OBJ = Counter(**'abrakadabra'**) print(OBJ) *# -> Counter({'a': 5, 'b': 2, 'r': 2, 'k': 1, 'd': 1})* OBJ = Counter({**'a'**: 5, **'b'**: 2, **'r'**: 2, **'k'**: 1, **'d'**: 1}) print(OBJ) *# -> Counter({'a': 5, 'b': 2, 'r': 2, 'k': 1, 'd': 1})* OBJ = Counter(python=2, java=4, ci=3) print(list(OBJ.elements())) *# -> ['python', 'python', 'java', 'java', 'java', 'java', 'ci', 'ci', 'ci']* print(Counter(**'abracadabra'**).most\_common(2)) *# -> [('a', 5), ('b', 2)]* print(Counter(**'abracadabra'**).most\_common()) *# -> [('a', 5), ('b', 2), ('r', 2), ('c', 1), ('d', 1)]* |

На практике одна из характерных областей, где применяется Counter – скрипты для обработки языка.

Типичные задачи:

1. Частотный анализ текстов
   1. Анализ наиболее/наименее частотных слов в тексте либо в корпусе текстов конкретного автора
   2. Сравнение наиболее/наименее частотной лексики у разных авторов или в разных областях знаний
2. Использование при [расчёте TF-IDF (оценки важности слова в контексте)](http://nlpx.net/tf-idf-%D1%81-%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D1%80%D0%B0%D0%BC%D0%B8-%D0%BA%D0%BE%D0%B4%D0%B0-%D0%BF%D1%80%D0%BE%D1%81%D1%82%D0%BE-%D0%B8-%D0%BF%D0%BE%D0%BD%D1%8F%D1%82%D0%BD%D0%BE/). TF для каждого слова рассчитывается как количество раз, когда слово встретилось в тексте, деленное на общее количество слов в тексте. Вот как-то так, например:

Counter (т.е. счетчик) в совокупности с регулярными выражениями позволяет выполнять частотный анализ текста.

**Листинг 2. task\_2.py**

|  |
| --- |
| **import** re **from** collections **import** Counter  find\_words = re.findall(**r'\w+'**, open(**'onegin.txt'**, encoding=**'utf-8'**).read().lower())  print(Counter(find\_words).most\_common(10)) *# -> [('и', 155), ('в', 68), ('не', 48), ('он', 45), ('я', 45), # ('на', 38), ('как', 32), ('но', 25), ('что', 25), ('с', 24)]* |

**Листинг 3. task\_3.py**

|  |
| --- |
| **import** re **from** collections **import** Counter  find\_words = re.findall(**r'\w+'**, open(**'mtsuri.txt'**, encoding=**'utf-8'**).read())   **def** tf\_calc(text):  *# преобразуем входной список в каунтер* tf\_text = Counter(text)  *# используем генератор словарей для деления значения каждого элемента  # в каунтере на общее число слов в тексте - т.е. длину списка слов.* tf\_text = {i: tf\_text[i] / float(len(text)) **for** i **in** tf\_text}  **return** tf\_text   print(tf\_calc(find\_words)) *# -> {'Мой': 0.0003048780487804878, 'дядя': 0.0006097560975609756, # 'самых': 0.0003048780487804878, 'честных': 0.0003048780487804878, 'правил': 0.0003048780487804878,* |

# DefaultDict (Словарь со значением по умолчанию)

Не совсем понятно, чем он отличается от обычного словаря. Давайте тогда сравним.

**Листинг 4. task\_4.py**

|  |
| --- |
| *"""Класс collections.defaultdict()"""* **from** collections **import** defaultdict  d = dict() d[**'раз'**] = 1 d[**'два'**] = 2 print(d) *# -> {'раз': 1, 'два': 2}* print(d[**'три'**]) *# -> KeyError: 'три' - ожидаемый результат* d = defaultdict(int) d[**'раз'**] = 1 d[**'два'**] = 2 print(d) *# -> defaultdict(<class 'int'>, {'раз': 1, 'два': 2})* print(d[**'три'**]) *# -> 0. Теперь ошибки нет* |

В первом случае результат ожидаемый. Мы обращаемся к ключу, которого нет, и соответственно получаем ошибку.

Если же необходимости получать исключение нет, можно воспользоваться альтернативой словаря – классом defaultdict. Его конструктору в виде аргумента передается тип данных элемента (в примере int). При этом конструктор поставит каждому элементу дефолтный элемент указанного типа, например, целого числа, строки, списка и т.д.

**Листинг 5. task\_5.py**

|  |
| --- |
| *"""Класс collections.defaultdict()"""* **from** collections **import** defaultdict  d = defaultdict(str) d[**'раз'**] = **'раз'** d[**'два'**] = **'два'** print(d[**'три'**]) *# -> defaultdict(<class 'str'>, {'раз': 'раз', 'два': 'два'})* dict\_of\_lst = defaultdict(list) **for** k **in** range(7):  dict\_of\_lst[k].append(k) print(dict\_of\_lst) *# -> defaultdict(<class 'list'>, {0: [0], 1: [1], 2: [2], 3: [3], 4: [4], 5: [5], 6: [6]})* |

Очень удобно, т.к. нет необходимости проверять наличие ключей или заранее создавать пустые списки.

Еще пример:

**Листинг 6. task\_6.py**

|  |
| --- |
| *"""Класс collections.defaultdict()"""* **from** collections **import** defaultdict  LST = [(**'yellow'**, 3), (**'blue'**, 2), (**'yellow'**, 3), (**'blue'**, 4), (**'red'**, 1)] DEF\_DICT = defaultdict(list) print(DEF\_DICT) **for** k, v **in** LST:  DEF\_DICT[k].append(v)  print(DEF\_DICT) print(DEF\_DICT.items()) print(sorted(DEF\_DICT.items())) print(DEF\_DICT[**'yellow'**]) |

Класс defaultdict, как и Counter, позволяет оптимизировать код. Рассмотрим пример:

**Листинг 7. task\_7.py**

|  |
| --- |
| *"""подсчет слов в предложении"""* **import** timeit **from** collections **import** defaultdict  SENTENCE = **"Ехал Грека через реку, Видит Грека — в реке рак "** \  **"Сунул Грека руку в реку — рак не цапает никак!"** WORDS = SENTENCE.split(**' '**)   **def** test\_simple\_dict():  *"""Обычный словарь"""* reg\_dict = {}  **for** word **in** WORDS:  **if** word **in** reg\_dict:  reg\_dict[word] += 1  **else**:  reg\_dict[word] = 1  **return** reg\_dict   **""" defaultdict автоматически назначает ноль как значение любому ключу,  который еще не имеет значения. Мы добавили одно, так что теперь в нем больше смысла, и оно также будет увеличиваться, если слово повторяется в  предложении несколько раз в предложении."""  def** test\_default\_dict():  *"""Вариант с defaultdict"""* d = defaultdict(int)  **for** word **in** WORDS:  d[word] += 1  **return** d  test\_default\_dict() |

Класс defaultdict автоматически назначит 0 как значение любому ключу, который даже еще не имеет значения.

# OrderedDict (Словарь с памятью порядка добавления элементов)

Если вы начали работать с Python, например, с версии 3.6, то наверняка не знаете, что словари в версиях до 3.6 были неупорядоченными (они «не помнили» порядка добавления элементов). Начиная с версии 3.6 словари стали более умными и «помнят» порядок добавления элементов.

**Листинг 8. task\_8.py**

|  |
| --- |
| *"""Класс collections.OrderedDict()"""* **import** collections  NEW\_DICT = {**'a'**: 1, **'b'**: 2, **'c'**: 3} *# -> с версии 3.6 порядок сохранится* print(NEW\_DICT) *# -> {'a': 1, 'b': 2, 'c': 3}  # а в версии 3.5 и более ранних можно было получить и такой результат # {'b': 2, 'c': 3, 'a': 1} # и вообще любой, ведь порядок ключей не сохранялся  # поэтому приходилось при необходимости обращаться к OrderedDict* NEW\_DICT = collections.OrderedDict([(**'a'**, 1), (**'b'**, 2), (**'c'**, 3)]) print(NEW\_DICT) *# -> OrderedDict([('a', 1), ('b', 2), ('c', 3)])* |

# Deque (Обобщение стеков и очередей)

Являются альтернативой спискам. Очень эффективно и быстро справляются с задачами добавления и извлечения некоторых объектов. Если же требуется случайный доступ, то быстрота здесь на стороне обычных списков.

Понятие Deque также расшифровывается как «Колода» или двусторонняя очередь. Это усовершенствованный список с оптимизацией вставок и удалений элементов с обоих концов. Операции слева и справа в этом классе имеют одинаковую производительность - O(1). Добавление в начало осуществляется быстрее, чем в обычных списках. Добавление в конец – не сильно уступает спискам.

**Листинг 9. task\_9.py**

|  |
| --- |
| *"""Класс collections.deque()""" # простые операции с очередью* **from** collections **import** deque  simple\_lst = list(**"bcd"**) deq\_obj = deque(simple\_lst) print(deq\_obj) *# -> deque(['b', 'c', 'd'])  # добавим элемент в конец очереди* deq\_obj.append(**'e'**) print(deq\_obj) *# -> deque(['b', 'c', 'd', 'e'])  # добавим элемент в начало очереди* deq\_obj.appendleft(**'a'**) print(deq\_obj) *# -> deque(['a', 'b', 'c', 'd', 'e'])  # pop также работает с обоих концов* deq\_obj.pop() deq\_obj.popleft() print(deq\_obj) *# -> deque(['b', 'c', 'd'])* |

# Еще пример с двусторонней очередью:

**Листинг 10. task\_10.py**

|  |
| --- |
| *"""Класс collections.deque()"""* **"""В соответствии с документацией Python,  deque – это обобщение стеков и очередей.  Вот основное правило: если вам нужно что-то быстро дописать или вытащить, используйте deque.  Если вам нужен быстрый случайный доступ, используйте list."""  import** string **from** collections **import** deque  *# формируем очередь из элементов-заглавных букв* NEW\_DEQUE = deque(string.ascii\_uppercase) print(NEW\_DEQUE)  *# итерируем очередь* **for** el **in** NEW\_DEQUE:  print(el, end=**' '**) print()  *# добавляем элемент в конец очереди* NEW\_DEQUE.append(**'end'**) print(NEW\_DEQUE)  *# добавляем элемент в начало очереди* NEW\_DEQUE.appendleft(**'start'**) print(NEW\_DEQUE)  *# перемещаем два элемента с конца очереди в начало* NEW\_DEQUE.rotate(2) print(NEW\_DEQUE)  *# перемещаем два элемента с начала очереди в конец* NEW\_DEQUE.rotate(-2) print(NEW\_DEQUE) |

# NamedTuple (Именованный кортеж)

В чем же его преимущество перед обычными кортежами? Прежде всего в том, что вам не нужно отслеживать индекс каждого элемента. Ведь у всех элементов есть названия, по которым к значениям элементов можно без труда обратиться. Здесь применяется нотация «обращение через точку», что очень выгодно, потому как представляете, если у вас будет 10 элементов и вам придется помнить позицию (индекс) каждого.

**Листинг 11. task\_11.py**

|  |
| --- |
| *"""Класс collections.namedtuple()"""* **from** collections **import** namedtuple  *# 'Resume' - имя кортежа # создаем шаблон кортежа* RES = namedtuple(**'Resume'**, **'id first\_name second\_name'**) print(RES) *# -> <class '\_\_main\_\_.Resume'> # заполняем шаблон данными* RESUME\_PARTS = RES(  id=**'1'**,  first\_name=**'Ivan'**,  second\_name=**'Ivanov'** )  print(RESUME\_PARTS) *# -> Resume(id='1', first\_name='Ivan', second\_name='Ivanov')* print(RESUME\_PARTS.id) *# -> 1* |

# ChainMap (Контейнер словарей)

Это специальный класс, умеющий из нескольких словарей создать надструктуру. Но это не один общий словарь, а их набор, и каждый словарь является независимым компонентом это надструктуры.

**Листинг 12. task\_12.py**

|  |
| --- |
| *"""ChainMap принимает любое количество сопоставлений или  словарей и превращать их в единое обновляемое представление."""* **from** collections **import** ChainMap  computer\_parts = {  **'system\_bock'**: 1,  **'monitor'**: 1,  **'keyboard\_mouse'**: 1 }  computer\_options = {  **'RAM'**: **'8 Gb'**,  **'HDD'**: **'1000 Gb'**,  **'PROC'**: **'Intel Core i5'** }  computer\_accessories = {  **'RAM'**: **'6 Gb'**,  **'gaming'**: **False**,  **'divided'**: **True**, }  **""" мы создали три словаря Python.  Далее, мы создали экземпляр ChainMap,  передав эти три словаря. В конце мы попытались  получить доступ к одному из ключей в нашем ChainMap. После этого, ChainMap пройдет через каждое сопоставление,  чтобы увидеть, существует ли данный ключ и имеет ли он значение.  Если это так, тогда ChainMap вернет первое найденное значение,  которое соответствует ключу."""** computer\_pricing = ChainMap(computer\_options, computer\_parts, computer\_accessories)  print(computer\_pricing) print(computer\_pricing[**'RAM'**])  computer\_pricing[**'RAM'**] = **'16 Gb'** print(computer\_pricing) |

Если вы обратитесь к ChainMap по ключу одного из словарей, то будет выполнен поиск значения среди всех словарей без необходимости указания конкретного словаря. При поиске ChainMap выведет первое обнаруженное значение, даже если в словарях несколько одинаковых ключей. ChainMap также поддерживает и операции изменения значений словарей.

**Выводы**

Counter – инструмент, осуществляющий подсчет неизменяемых объектов. Позволяет узнать число вхождений или наиболее (наименее) часто встречающихся элементов.

DefaultDict – словарь, который при обращении к несуществующему ключу вместо возврата исключения записывает значение по умолчанию. (отрабатывает быстрее чем использование setdefault() у dict()).

OrderedDict – словарь, который «помнит» порядок добавления элементов.

ChainMap – контейнер словарей с обобщением ключей и элементов.

NamedTuple – фабрика кортежей, позволяет вместо индексов использовать имена для обращения к элементам.

Deque – двусторонняя очередь («колода») – специализированный список, оптимизированный для операций вставки и удаления элементов с обоих концов.

<https://proglib.io/p/ne-izobretat-velosiped-ili-obzor-modulya-collections-v-python-2019-12-15>

<https://nuancesprog.ru/p/3295/>

<https://codecamp.ru/documentation/python/498/collections-module>

<https://python-scripts.com/import-collections>