

Урок 5. Специализированные коллекции

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 2](#_Toc88659101)

[ВВЕДЕНИЕ 3](#_Toc88659102)

[COUNTER (СЧЕТЧИК) 4](#_Toc88659103)

[DEFAULTDICT (СЛОВАРЬ СО ЗНАЧЕНИЕМ ПО УМОЛЧАНИЮ) 6](#_Toc88659104)

[ORDEREDDICT (СЛОВАРЬ, «ЗАПОМИНАЮЩИЙ» ПОРЯДОК ДОБАВЛЕНИЯ ЭЛЕМЕНТОВ) 10](#_Toc88659105)

[NAMEDTUPLE (ИМЕНОВАННЫЙ КОРТЕЖ) 12](#_Toc88659106)

[CHAINMAP (КОНТЕЙНЕР СЛОВАРЕЙ) 13](#_Toc88659107)

[ВЫВОДЫ 14](#_Toc88659108)



# ВВЕДЕНИЕ

Вы уже хорошо знакомы из курса основ Python со встроенными контейнерами данных (встроенными коллекциями), такими, как списки, словари, 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. поддерживают интерфейс стандартных коллекций и предоставляют разработчику еще некоторый дополнительный функционал. Плюс некоторые операции с **collections** выполняются быстрее, что для нашего курса очень актуально.

Т.е. collections могут сделать решение более лаконичным и быстрым в плане исполнения. Работу с **collections** можно еще охарактеризовать фразой «не изобретать велосипед». Здесь имеется в виду, что решение можно построить и на обычных коллекциях, но оно может быть довольно объемным.

**Collections** это не фреймворк, а лишь некоторый дополнительный инструментарий разработчика. Модуль **collections** не определяет архитектуру проекта, как, например, фреймворк Django.

ВЫВОД: какая выгода их использования **collections**? Оптимизация кода! Вы можете решить задачу и без специализированных коллекций, но объем кода будет больше, чем если бы вы воспользовались **collections**. Это именно то, что нужно нам в этом курсе – изучить как можно больше средств оптимизации алгоритмов. Модуль **collections** является встроенным компонентом стандартной библиотеки Python.

Такие модули, как **collections**, именуются «батарейками», т.е. это компоненты, в которых уже существуют встроенные возможности решения некоторых стандартных задач, например, частотный анализ слов и т.д. Стоит также отметить, что каждый класс-конструктор из модуля **collections** наследует интерфейс какой-то традиционной встроенной коллекции.

# COUNTER (СЧЕТЧИК)

Одной из стандартных задач, когда разработчики придумывают свои собственные решения, является частотный анализ – подсчет элементов последовательностей (списков, строк и т.д.). Например, у нас есть файл с данными о работе компании за некоторый период. Это могут быть продажи товаров по дням за год. Мы хотим найти самые продаваемые товары или посмотреть какие количества продаж были самые частые.

В обычном случае вам придется написать свое решение на базе цикла 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 класса-конструктора **Counter**.

Он принимает итерируемый объект и возвращает словарь, где ключами являются элементы объекта, а значениями – частоты каждого элемента. **Counter** наследует интерфейс словаря.

**Листинг 1. counter\_ex/task\_1.py**

|  |
| --- |
| *"""Класс collections.Counter()"""* **from** collections **import** Counter  *# создаем объект коллекции* OBJ = Counter([**'js'**, **'java'**, **'java'**, **'python'**, **'python'**, **'python'**]) print(type(OBJ)) 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)]* |

Практике одна из характерных областей, где применяется **Counter** – скрипты для обработки данных, парсеры, аналитические приложения и т.д.

Еще примеры задач:

1. Частотный анализ текстов. Анализ наиболее/наименее частотных слов в тексте либо в корпусе текстов конкретного автора.
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.

Поразмышляйте, какую из практических задач урока 4 вы могли решить буквально в одну строку средствами класса **Counter**?

**Листинг 2. counter\_ex/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. counter\_ex/task\_3.py**

|  |
| --- |
| *# SEO* **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  # показатель важности слова в контексте* 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] / 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. defaultdict\_ex/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* print(d) |

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 параметр в качестве значения, в данном случае, целого числа (по дефолту будет значение 0). Также можно указать и другие типы данных, например, строку, список и т.д. Будет определено соответствующее дефолтное значение (пустая строка, список и т.д.).

Вы скажете, что в обычных словарях есть метод **setdefault()**, который умеет делать то же самое. Но возможно они различаются по скорости выполнения? На разных ресурсах можно встретить различные оценки производительности этих вариантов. Кто прав?

Проведем замеры.

**Листинг 5. defaultdict\_ex/task\_5.py**

|  |
| --- |
| *"""Класс collections.defaultdict()"""* **from** collections **import** defaultdict **from** timeit **import** timeit   **def** check\_1():  dict\_of\_lst = defaultdict(list)  dict\_of\_lst[**"model"**]  **return** dict\_of\_lst   **def** check\_2():  dict\_of\_lst = dict()  dict\_of\_lst.setdefault(**"model"**, list())  **return** dict\_of\_lst   print(check\_1()) *# -> defaultdict(<class 'list'>, {'model': []})* print(check\_2()) *# -> {'model': []}* print(  **'defaultdict: '**,  timeit(  **'check\_1()'**,  globals=globals())) print(  **'setdefault: '**,  timeit(  **'check\_2()'**,  globals=globals()))   **""" defaultdict: 0.41834081100000003**  **setdefault: 0.35510432700000005 """** |

По приведенным цифрам можно предположить, что **defaultdict** проиграл, но так ли это на самом деле?

По сути, что мы сделали в приведенном листинге? Создали новый объект **defaultdict**. Эта операция вполне может быть выполнять дольше, чем аналогичная операция в **dict**. Но создаем мы один раз, а применять можем многократно.

Поэтому наши замеры нужно перенести на некоторое количество итераций и провести замеры там.

**Листинг 6. defaultdict\_ex/task\_6.py**

|  |
| --- |
| **from** timeit **import** timeit **from** collections **import** defaultdict  s = [(**'раз'**, 1), (**'два'**, 2), (**'раз'**, 3), (**'два'**, 4), (**'три'**, 1)] d\_1 = defaultdict(list) d\_2 = {}   **def** check\_1():  **for** k, v **in** s:  d\_1[k].append(v)  **return** d\_1   **def** check\_2():  **for** k, v **in** s:  d\_2.setdefault(k, []).append(v)  **return** d\_2   print(timeit(check\_1)) *# 0.78448056* print(timeit(check\_2)) *# 1.2408615469999997* |

Замеры показывают, что операции **defaultdict** выполняются быстрее.

С чем связаны такие цифры?

Дело в том, что **defaultdict** быстрее, чем **dict.setdefault()**, поскольку первый определяет дефолтное значение для всего словаря при его создании, а второй выполняет это для каждого элемента при его чтении.

Кроме того, класс **defaultdict**, как и **Counter**, позволяет сделать код более лаконичным. Рассмотрим пример:

**Листинг 7. defaultdict\_ex/task\_7.py**

|  |
| --- |
| *"""подсчет слов в предложении"""* **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 |

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

# ORDEREDDICT (СЛОВАРЬ, «ЗАПОМИНАЮЩИЙ» ПОРЯДОК 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. ordereddict\_ex/task\_8.py**

|  |
| --- |
| **from** collections **import** OrderedDict  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 = OrderedDict([(**'a'**, 1), (**'b'**, 2), (**'c'**, 3)]) print(NEW\_DICT[**'a'**]) *# -> OrderedDict([('a', 1), ('b', 2), ('c', 3)])* |

Спецификация Python 3.7 подтверждает, что встроенные словари упорядочены, т.е. элементы размещаются в том порядке, в котором они были добавлены. это делает классы dict и ordereddict фактически одинаковыми.

При этом на некоторых ресурсах есть информация, что операции с **ordereddict** выполняются быстрее, нежели с **dict**. это мы проверим в рамках выполнения домашнего практического задания.

итак, получается необходимости в **ordereddict** скорее всего больше нет? на самом деле, некоторые моменты указывают на то, что упорядоченные словари сохраняют свою актуальность.

* Указание на важность порядка элементов. При использовании **dict** мы полагаемся на базовые возможности словаря. В случае с **ordereddict** явно показываем, что нам важен именно порядок следования элементов. Т.е. это пункт на уровне протокола, соглашения.

**Листинг 9. ordereddict\_ex/task\_9.py**

|  |
| --- |
| **from** collections **import** OrderedDict  *# порядок* NEW\_DICT = {**'a'**: 1, **'b'**: 2, **'c'**: 3} print(NEW\_DICT) *# -> {'a': 1, 'b': 2, 'c': 3}* NEW\_DICT = OrderedDict([(**'a'**, 1), (**'b'**, 2), (**'c'**, 3)]) print(NEW\_DICT) *# -> OrderedDict([('a', 1), ('b', 2), ('c', 3)])* |

* Возможность оперативного переупорядочивания элементов. У **ordereddict** существует специальный метод **.popitem()**. В случае с обычным словарем такой оперативности при переупорядочивании не будет, поскольку потребуется время на удаление и повторную установку элемента.

**Листинг 10. ordereddict\_ex/task\_10.py**

|  |
| --- |
| **from** collections **import** OrderedDict  *# специальные возможности* NEW\_DICT = OrderedDict([(**'a'**, 1), (**'b'**, 2), (**'c'**, 3)]) print(NEW\_DICT) *# переносит ключ переносит элемент с указанным в конец, # если last=True, и в начало, если last=False* NEW\_DICT.move\_to\_end(**'b'**, last=**True**) print(NEW\_DICT) *# удаляет последний элемент если last=True, и первый, если last=False* res = NEW\_DICT.popitem(last=**True**) print(res) *# ('b', 2)* print(NEW\_DICT) *# OrderedDict([('a', 1), ('c', 3)])* |

* Проверка словарей на эквивалентность (равенство). Причем важен именно порядок элементов.

**Листинг 11. ordereddict\_ex/task\_11.py**

|  |
| --- |
| **from** collections **import** OrderedDict  *# специальные возможности* NEW\_DICT = OrderedDict([(**'a'**, 1), (**'b'**, 2), (**'c'**, 3)])  NEW\_DICT\_0 = OrderedDict(a=1, b=2, c=3, d=4) NEW\_DICT\_1 = OrderedDict(b=2, a=1, c=3, d=4) NEW\_DICT\_2 = OrderedDict(a=1, b=2, c=3, d=4)  print(NEW\_DICT\_0 == NEW\_DICT\_1) *# False* print(NEW\_DICT\_0 == NEW\_DICT\_2) *# True* NEW\_DICT\_0 = dict(a=1, b=2, c=3, d=4) NEW\_DICT\_1 = dict(b=2, a=1, c=3, d=4) NEW\_DICT\_2 = dict(a=1, b=2, c=3, d=4)  print(NEW\_DICT\_0 == NEW\_DICT\_1) *# True* print(NEW\_DICT\_0 == NEW\_DICT\_2) *# True* print(NEW\_DICT\_0 == NEW\_DICT\_1 == NEW\_DICT\_2) *# True* NEW\_DICT\_0 = OrderedDict(a=1, b=2, c=3, d=4) NEW\_DICT\_1 = dict(b=2, a=1, c=3, d=4)  print(NEW\_DICT\_0 == NEW\_DICT\_1) *# True* |

# 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.

Подход работы с именованными кортежами чем-то напоминает принцип работы с объектами пользовательских классов, когда обращение к атрибутам и методам осуществляется через точку. Именованные кортежи сохраняют все плюсы обычных кортежей – защищены от изменений, требуют меньшей памяти, при этом обращение к элементам осуществляется по имени и чем-то напоминает работу с элементами словаря.

**Листинг 14. namedtuple\_ex/task\_14.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* |

По сути, функция namedtuple представляет собой фабрику для создания именованных кортежей. Стоит отметить, что, хотя кортеж является именованным, он сохраняет обращение к элементам по индексам.

# CHAINMAP (КОНТЕЙНЕР СЛОВАРЕЙ)

Это специальный класс, умеющий для нескольких классов создать контейнер. Работая с контейнером словарей, мы работаем как будто с одним большим словарем, но на самом деле контейнер лишь создает видимость такой работы. Все классы остаются независимыми.

Если нам нужно будет обратиться к элементу одного из словарей по ключу, достаточно будет указать лишь имя контейнера. Имя словаря указывать не требуется.

**Листинг 15. chainmap\_ex/task\_15.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**, }  lst = [computer\_parts, computer\_options, computer\_accessories] lst[0][**'monitor'**]    **""" 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)  *# lst[0]['monitor']* print(computer\_pricing[**'RAM'**])  computer\_pricing[**'opt'**] = **'20 Gb'** computer\_pricing[**'RAM'**] = **'1000 Gb'** print(computer\_options) |

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, при этом опять же изменение будет выполняться для первого совпадения. **ChainMap** поддерживает такие операции со словарями, как **keys()** и **values()**.

Применение **ChainMap** отмечено в таких проектах, как парсеры и телеграм-боты.

# 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.

Мы не обсудили еще один важный компонент, имеющий отношение к модулю **collections**. Это наследованный модуль коллекции абстрактных базовых классов **collections.abc**.

О нем мы плотно поговорим на одном из следующих курсов. Его задача – контролировать интерфейс классов-наследников, т.е. обеспечивать проверку наличия определенных методов у наследников.

**Листинг 16. chainmap\_ex/task\_16.py**

|  |
| --- |
| **import** abc   **class** PaymentService(abc.ABC):  @abc.abstractmethod  **def** paypal(self):  **pass   class** Site(PaymentService):  **def** paypal(self):  **pass** obj = Site() obj.paypal() |

В данном случае если мы забудем переопределить указанный метод у класса-потомка, мы получим исключение:

TypeError: Can't instantiate abstract class Site with abstract methods paypal

Можно сказать, что модуль **abc** не несет какой-то большой смысловой нагрузки. Его значимость второстепенна и больше касается каких-то вещей на уровне соглашений. Тем не менее, его использование относится к «хорошему» стилю программирования.