**Модуль 8. Стандартные абстрактные типы данных  (4 ак. ч.)**

* Контейнерные типы.
* Библиотечные модули collections и collections.abc

Практикум. Схема «Компонент-Контейнер» и ее применение

Оглавление

[Контейнерные типы 1](#_Toc147169824)

[**Немного про особенности всех трёх** 4](#_Toc147169825)

[Абстрактные структуры данных: 4](#_Toc147169826)

[Стек 4](#_Toc147169827)

[Практикум по стеку 11](#_Toc147169828)

[Очередь 14](#_Toc147169829)

[Библиотечные модули collections и collections.abc 16](#_Toc147169830)

[Counter 16](#_Toc147169831)

[Словарь со значением по умолчанию (defaultdict) 20](#_Toc147169832)

[Двусторонняя очередь (deque) 25](#_Toc147169833)

[Именованный кортеж и функция namedtuple() 28](#_Toc147169834)

[Модуль collections.abc 34](#_Toc147169835)

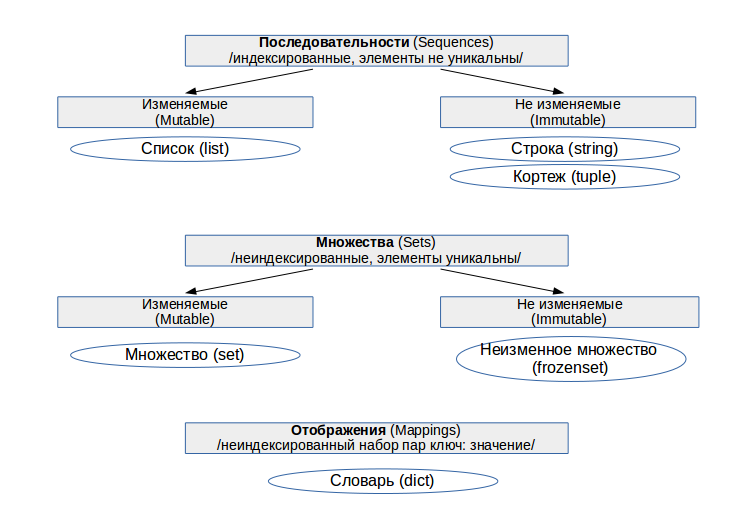
## Контейнерные типы

Контейнерами ( переменными-контейнерами) называют типы, которые умеют содержать в себе другие типы (часто их еще называют «коллекциями»), а также позволяющие обращаться к этим значениям, а также применять специальные функции и методы, зависящие от типа коллекции.

Частая проблема при изучении коллекций заключается в том, что разобрав каждый тип довольно детально, обычно потом не уделяется достаточного внимания разъяснению картины в целом, не проводятся чёткие сходства и различия между типами, не показывается как одну и туже задачу решать для каждой из коллекций в сравнении.

Вот именно эту проблему мы попытаемся решить в данном курсе – рассмотреть ряд подходов к работе со стандартными коллекциями в Python в сравнении между коллекциями разных типов, а не по отдельности, как обычно в обучающих материалах. Кроме того, постараюсь затронуть некоторые моменты, вызывающие сложности и ошибки у начинающих.

**Будем рассматривать**стандартные встроенные коллекционные типы данных в Python: список (list), кортеж (tuple), строку (string), множества (set, frozenset), словарь (dict). Коллекции из модуля collections рассматриваться не будут, хотя многое из статьи должно быть применимым и при работе с ними.



**Индексированность** – каждый элемент коллекции имеет свой порядковый номер — индекс. Это позволяет обращаться к элементу по его порядковому индексу, проводить слайсинг («нарезку») — брать часть коллекции выбирая исходя из их индекса. Детально эти вопросы будут рассмотрены в дальнейшем в отдельной статье.  
  
**Уникальность** – каждый элемент коллекции может встречаться в ней только один раз. Это порождает требование неизменности используемых типов данных для каждого элемента, например, таким элементом не может быть список.  
  
**Изменяемость** коллекции — позволяет добавлять в коллекцию новых членов или удалять их после создания коллекции.



#### **Примечание для словаря (dict):**

* сам словарь изменяем — можно добавлять/удалять новые пары ключ: значение;
* значения элементов словаря — изменяемые и не уникальные;
* а вот ключи — не изменяемые и уникальные, поэтому, например, мы не можем сделать ключом словаря список, но можем кортеж. Из уникальности ключей, так же следует уникальность элементов словаря — пар ключ: значение.

UPD: Важное замечание : Для того, чтобы объект мог быть ключом словаря, он должен быть хешируем. У кортежа, возможен случай, когда его элемент является не хешируемым объектом, и соответственно сам кортеж тогда тоже не является хешируемым и не может выступать ключом словаря.

a = (1, [2, 3], 4)

print(type(a)) # <type 'tuple'>

b = {a: 1} # TypeError: unhashable type: 'list'

Нас сейчас интересуют три таких сущности, с которыми мы начнем знакомиться в этом блоке:

list — список. В Python список — это упорядоченная коллекция объектов, при этом сами объекты не обязаны быть одного и того же типа. В Phyton списки создаются через квадратные скобки [ ].

tuple — кортеж. Эти ребята, по своей сути, почти ничем не отличаются от списков, кроме того, что их компоненты нельзя изменять после создания. Таплы создаются с помощью круглых скобок ( ).

dict — словарь. Словарь — это структура, которая дает возможность связывать пары ключ-значение и достать значение по ключу. Наверное, самая непростая для понимания из всех трех. В Python за словарями закреплены фигурные скобки { }.

**Немного про особенности всех трёх**

У всех этих трех типов есть свои нюансы, которые мы подробнее разберем позднее. Но, например, за счет того, что элементы tuple нельзя изменять, интерпретатор всегда знает, сколько они занимают места в памяти и как с ними работать. Иногда это может повлиять на скорость выполнения (в лучшую сторону). У таплов еще есть несколько удобных особенностей, которые мы подробнее рассмотрим позже. Списки гарантируют порядок элементов, которые они в себе хранят. При этом часто таплы и списки взаимозаменяемы (например, и те, и другие имеют механизм для извлечения только части последовательности). А словари позволяют строить сложные зависимости и предоставляют механизм их обхода. За счет особенности хранения, словари позволяют быстро получать необходимые элементы по ключу.

## Абстрактные структуры данных:

### Стек

На собеседовании вам вполне могут предложить написать код для реализации стека или очереди. Да, на работе вам, вероятно, за всю карьеру не случится этого делать, но задачи для вайтбоардинга отличаются своей нежизненностью. Несмотря на это, такие задачи — отличный способ продемонстрировать, что вы умеете реализовывать классы (т. е. создавать вещи, которые делают то, что вы сказали им делать).



Концепции стеков и очередей довольно простые. Стек (англ. stack — стопка) похож на стопку тарелок. Если у вас есть такая стопка, убирать тарелки вы будете, начиная сверху. Таким образом, последняя тарелка, которую вы положили на стопку, будет убрана первой. Вероятно, вы слышали термин LIFO — Last In First Out («последним пришел, первым ушел»).

А в очередях все наоборот: первый элемент в очереди удаляется тоже первым. Представьте себе очередь в Starbucks. Первый человек в очереди всегда получает свой кофе первым.

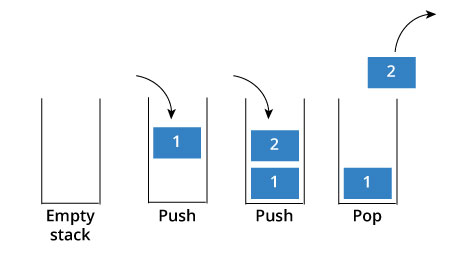
Всё работает на стеках. Процессор имеет свой стек. i5

Вызов функции тоже кладется на стек. Рекурсии помещаются в стек, а потом выбрасываются.

В жизни мы живем в стеке. Когда начальник подбрасывает задач, предыдущая уходит в стек, звонит телефон и т.д.

Браузер – история посещений, кнопка вперед-назад, кругом стековая архитектура.

Принцип Last-In-First-Out (LIFO). Пример.



Что мешает нам работать со списком, как со стеком. Append – есть, pop – есть!

Нам не нужно залезать в середину, в основание и т.п. Нам не нужно ничего искать по значению или точечно – по индексу!

Но нам не нужны для стека все примочки. Не будем вносить искусственные ограничения в класс List и блокировать ненужные нам методы

Напишем свой стек. Также как и писали двусвязанный список.

Стек имеет свой интерфейс. Какой?

*# Тз для стека  
# push(item)  
# pop()  
# peek() посмотреть   
# is\_empty  
# size*

Пример. Проверка палиндрома

def is\_palindrome(word):  
 word = word.lower()  
 rword = ""  
  
 stack = Stack()  
  
 for char in word:  
 stack.push(char)  
  
 while not stack.is\_empty():  
 rword += stack.pop()  
  
 return word == rword

Мы знаем 3 функции для проверки палиндрома, а вот еще одна – с использованием стека.

Мы принимаем слово, пушим (закидываем) в стек. А потом обратно вытащим её из стека.

Получим строку задом-наперед.

Опишем стек.

*# Тз для стека  
# push(item)  
# pop()  
# peek() посмотреть  
# is\_empty  
# size*class Stack:  
 def \_\_init\_\_(self):  
 self.\_\_data = list()  
  
 def push(self, item):  
 self.\_\_data.append(item)  
  
 def pop(self):  
 if len(self.\_\_data) > 0:  
 return self.\_\_data.pop()  
 return None  
  
 def peek(self):  
 if len(self.\_\_data) > 0:  
 return self.\_\_data[-1]  
  
 def is\_empty(self):  
 return len(self.\_\_data)==0  
  
 def show(self):  
 print('\n'.join(str(val) for val in self.\_\_data))

Результат

s = Stack()

s.is\_empty()

True

s.push(10)

s.is\_empty()

False

s.push(20)

s.show()

10

20

s.peek()

20

s.pop()

20

s.pop()

10

Проверим палиндром

print(is\_palindrome('шалаш')) # True

тест

А роза упала на лапу Азора

Как реализовать?

Replace(“ “,””)

Как интепретатор парсит наш код?

Как он правильно находит парные скобки.

Это работа для стека. А

( ( ) ( ( ) ) )

Строка читается слева направо. Как только мы встречаем строку, мы кладем ее на стек

(

Далее снова открывающаяся скобка, снова кладем ее на стек

( (

Как только встретили закрывающуюся скобку, выпиливаем предыдущую открывающуюся из стека

(

И так далее

(((

((

(

\_

Всё. Стек пустой. Если стек не пустой – ошибка интерпретатора.

Пишем свой парсер (скобок)

from stack import Stack  
  
  
def brackets\_checker(s):  
 stack = Stack()  
  
 for char in s:  
 if char == '(':  
 stack.push(char)  
 else:  
 if stack.is\_empty():  
 return False  
 else:  
 stack.pop()  
 if stack.is\_empty():  
 return True  
 return False  
  
  
print(brackets\_checker('(()())')) #True

Работает!

Но скобки бывает фигурные, квадратные.

Как нашу функцию расширить для проверки всех скобок?

# ([](){())}

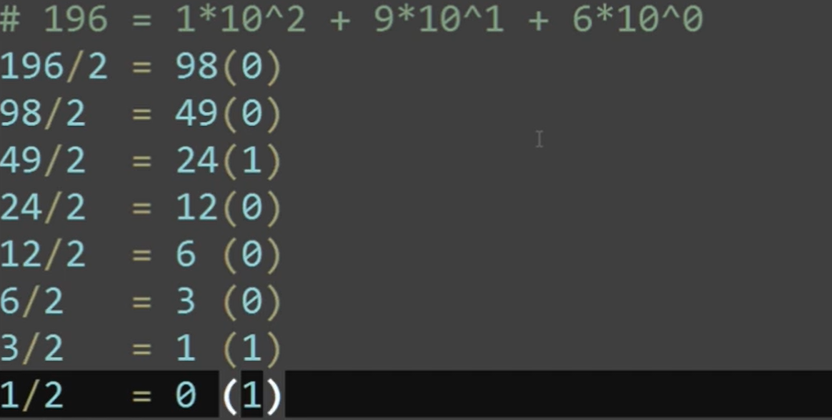
Чтобы вы сделали?

Стек для каждой скобки?

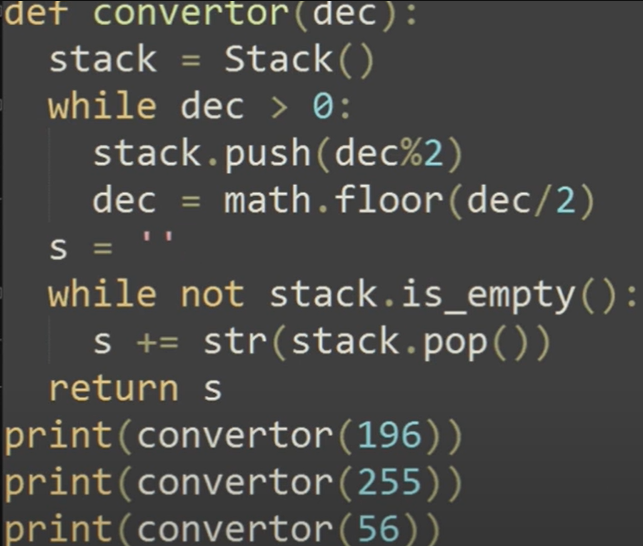
Или еще вариант решения

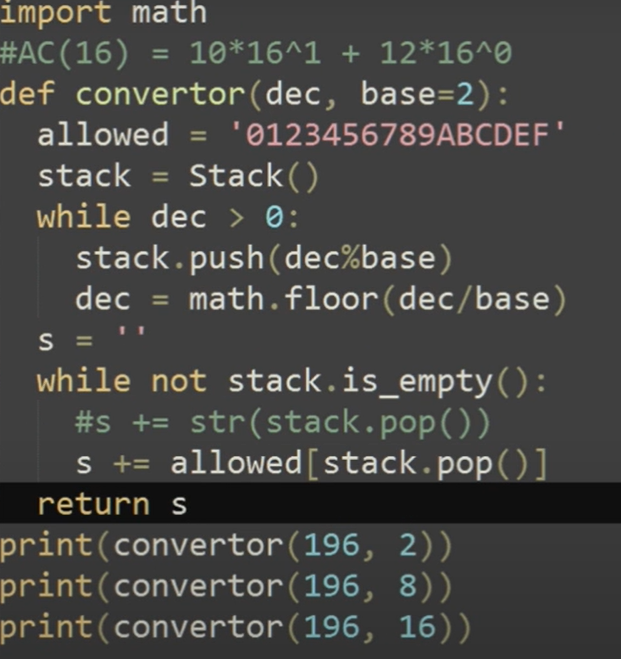
o\_allow = '([{'  
c\_allow = ')]}'

Почувствует себя в шкуре разработчиков интепретаторов или парсеров.



С помощью стека





String hexdigits

Понимание рекурсии

На примере факториала. Каждая функция кладется на стек.

И обратно, циклом вытягиваем оттуда значение и перемножаем одно на другое.

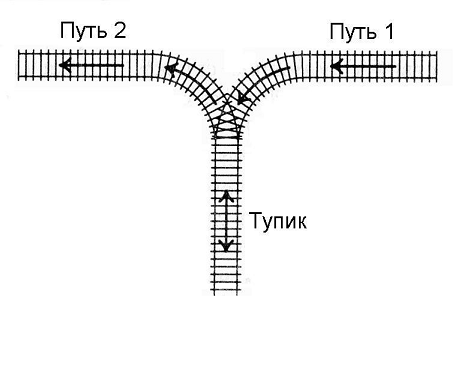
Хорошая рекурсия, вместо функции кладем просто значения.

### Практикум по стеку

Сортировка вагонов

Олимпиадная задача для старшеклассников.

Для буржуйских стран это вообще не понятно, как так можно разворачиваться, но у нас мало пространства в стране, особенно за уралом (шутка)



К тупику со стороны пути 1 подъехал поезд. Необходимо сделать так, чтобы вагоны поезда шли по порядку попали на путь 2 по порядку (сначала первый, потом второй и т.д., считая от головы поезда, едущего по пути 2 в сторону от тупика).

При этом известно, в каком порядке изначально идут вагоны поезда.

* с пути 1 попасть на путь 2, не заезжая в тупик, нельзя.
* разрешается отцепить от поезда один или сразу несколько первых вагонов и завезти их в тупик
* часть вагонов из тупика можно вывезти на путь 2
* первые две операции можно повторять. При этом:
* каждый вагон может лишь один раз заехать с пути 1 в тупик, а затем один раз выехать из тупика на путь 2
* заезжать в тупик с пути 2 или выезжать из тупика на путь 1 запрещается
* Формат входных данных
  + вводятся числа через пробел - номера вагонов в порядке от головы поезда, едущего по пути 1 в сторону тупика. Вагоны пронумерованы натуральными числами от 1, каждое из которых встречается ровно один раз.
* Формат выходных данных
  + напечатать строку "Вывели", если все вагоны удалось перегнать с пути 1 на путь 2
  + напечатать строку "Не получилось", если перегон вагонов невозможен

Как это должно работать

cmd

>>> 3 2 1

Вывели

>>> 4 1 3 2

Вывели

>>> 2 3 1

Не получилось

Решение

from stack import Stack  
  
  
def do\_lab(train):  
 *# '3 2 1'* dock = Stack()  
 train = train.split() *# ['3', '2', '1']* train = list(map(int, train)) *# [3, 2, 1]* needed = 1  
 for car in train:  
 dock.push(car)  
 if car == needed:  
 dock.pop()  
 needed += 1  
 while dock.size() > 0:  
 if dock.peek() == needed:  
 dock.pop()  
 needed += 1  
 else:  
 break  
 if dock.is\_empty():  
 print('Вывели')  
 else:  
 print('Не вывели')  
  
  
while True:  
 nums = input('Введите номер вагонов : ')  
 if nums == '0':  
 break  
 do\_lab(nums)

Это решение можно использовать не только для паровозов. Классика стек

Технически можно было оба пути сделать стеками.

Но, это облегченное задание.

В реальном задании нужно было написать количество вводимых-выводимых поездов.

Ввели 1

Вывели 1

-> 3 2 1

Ввели 3

Вывели 3

-> 4 1 3 2

Ввели 2

Вывели 1

Ввели 2

Вывели 3

Сколько телодвижений потребовалось.

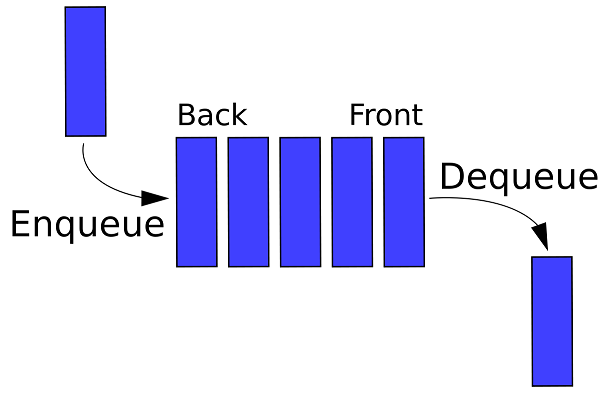
### Очередь

Очередь в Python – это линейный тип структуры данных, используемый для последовательного хранения данных. Ее концепция основана на FIFO (“First in First Out”), что означает «первым пришел – первым обслужен». Или так: «первым пришел – первый вышел». Очередь имеет два конца – спереди и сзади. Следующий элемент вставляется с заднего конца и снимается с переднего конца.



Например, в лаборатории информатики 20 компьютеров, подключенных к одному принтеру. Студенты хотят напечатать свою работу; принтер напечатает первое задание, второе и так далее. Если мы будем последними, нам нужно дождаться завершения всех других задач, которые опережают нашу.

Операционная система управляет очередью для обработки различных процессов на компьютере.



Мы можем выполнять следующие операции в очереди.

Enqueue – постановка в очередь – операция, которая добавляет элементы. Является условием, если очередь заполнена. Временная сложность постановки составляет O (1).

Dequeue – это операция, при которой мы удаляем элемент из очереди. Он удаляется в том же порядке, в котором был вставлен. Если очередь пуста, это является условием потери значимости. Временная сложность составляет O (1).

Front – элемент вставлен в переднюю часть. Временная сложность – O (1).

Rear – элемент удален из задней части. Временная сложность задней части O (1).

Доступные методы в очереди

Python предоставляет следующие методы, которые обычно используются для выполнения операции в очереди:

Принцип First-In-First-Out (FIFO)[¶](file:///C:\Users\User\Desktop\%D0%A1%D0%9F%D0%95%D0%A6%D0%98%D0%90%D0%9B%D0%98%D0%A1%D0%A2\PYTHON_2\%D0%9C%D0%90%D0%A2%D0%95%D0%A0%D0%98%D0%90%D0%9B%D0%AB\docs\html\11-module-4-queue.html#Принцип-First-In-First-Out-(FIFO))

put(item) – эта функция используется для вставки элемента.

get() – используется для извлечения элемента.

empty() – эта функция предназначена для проверки, пуста ли очередь или нет. Если пуста – она возвращает истину.

qsize – возвращает длину очереди.

full() – при условии, если очередь заполнена, возвращает истинное значение; в противном случае -ложное.

Изучим эти функции в следующих разделах.

class Queue:  
  
 def \_\_init\_\_(self):  
 self.\_\_data = list()  
  
 def enqueue(self, item): *# ставить в очередь* self.\_\_data.append(item)  
  
 def dequeue(self): *# брать из очереди* if len(self.\_\_data) > 0:  
 return self.\_\_data.pop(0)  
 return None  
  
 def rear(self):  
 if len(self.\_\_data) > 0:  
 return self.\_\_data[-1]  
 return None  
  
 def front(self):  
 if len(self.\_\_data) > 0:  
 return self.\_\_data[0]  
 return None  
  
 def is\_empty(self):  
 return len(self.\_\_data) == 0  
  
 def show(self):  
 print('\n'.join(str(val) for val in self.\_\_data))  
  
 def size(self):  
 return len(self.\_\_data)  
  
 def cleat(self):  
 self.\_\_data = list()

Хрестоматийный пример: игра Hot Potato



Hot potato - это простая игра, которая развивает зрительно-моторную координацию и командную работу.

Чтобы сыграть в горячую картошку, игроки пасут мяч по кругу (под музыку или под счет). Когда музыка прекращается (или счет достигает заданной цифры), тот, кто держит мяч, оказывается вне игры!

Предположим, что у нас есть список имен. И есть какое-то число.

А для симуляции очень удобно использовать очередь.

У кого в руке картошка – тот первый в очереди.

После того, как он ее бросил, мы извлекаем его из очереди. И ставим его обратно в хвост.

И так они будут извлекаться и ставиться в очередь

def hot\_potato(names, num):  
 queue = Queue()  
  
 for name in names:  
 queue.enqueue(name)  
  
 while queue.size() > 1:  
 for i in range(num):  
 queue.enqueue(queue.dequeue())  
  
 eliminated = queue.dequeue()  
 print(f'{eliminated} вышел из игры')  
  
 return queue.dequeue()  
  
names = ['Вася', 'Петя', 'Зина', 'Федя']  
winner = hot\_potato(names, 3)  
print(winner)

Можно подредактить код

class Queue:  
  
 def \_\_init\_\_(self):  
 self.\_\_data = list()  
  
 def enqueue(self, item): *# ставить в очередь* self.\_\_data.append(item)  
  
 def dequeue(self): *# брать из очереди* if len(self.\_\_data) > 0:  
 return self.\_\_data.pop(0)  
 return None  
  
 def rear(self):  
 if len(self.\_\_data) > 0:  
 return self.\_\_data[-1]  
 return None  
  
 def front(self):  
 if len(self.\_\_data) > 0:  
 return self.\_\_data[0]  
 return None  
  
 def is\_empty(self):  
 return len(self.\_\_data) == 0  
  
 def show(self):  
 print(', '.join(str(val) for val in self.\_\_data))  
  
 def size(self):  
 return len(self.\_\_data)  
  
 def cleat(self):  
 self.\_\_data = list()  
  
  
def hot\_potato(names, num):  
 queue = Queue()  
  
 for name in names:  
 queue.enqueue(name)  
  
 while queue.size() > 1:  
 for i in range(num):  
 queue.enqueue(queue.dequeue())  
  
 queue.show()  
 eliminated = queue.dequeue()  
 print(f'{eliminated} вышел из игры')  
  
 return queue.dequeue()  
  
  
names = ['Вася', 'Петя', 'Зина', 'Федя']  
winner = hot\_potato(names, 3)  
print(winner)

Результат:

Федя, Вася, Петя, Зина

Федя вышел из игры

Вася, Петя, Зина

Вася вышел из игры

Зина, Петя

Зина вышел из игры

Петя

Классика жанра, хрестоматийный пример, который разбирается везде.

Еще один пример – олимпийская задача.

Идея – у нас есть список ученик (в файле).

В каждой строке список такого типа

lst = [  
 '9 Иванов',  
 '10 Петров',  
 '11 Сидоров',  
 '9 Морозов',  
 '9 Беляев',  
 '10 Борисов'  
]

Сначала нужно вывести 9-классников, потом 10, потом 11-классников.

Именно в таком порядке. Решается с помощью очередей.

Для 9-го класса своя очередь, для 10-го своя и т.д.

def make\_list\_of\_students(lst):  
 q9,q10,q11 = Queue(),Queue(),Queue()  
  
 for item in lst:  
 student = item.split()  
 if student[0] == '9':  
 q9.enqueue(student[1])  
 elif student[0] == '10':  
 q10.enqueue(student[1])  
 else:  
 q11.enqueue(student[1])  
 while q9.size() > 0:  
 print('9 ', q9.dequeue())  
  
 while q10.size() > 0:  
 print('10 ', q10.dequeue())  
  
 while q11.size() > 0:  
 print('11 ', q11.dequeue())  
  
  
  
  
lst = ['9 Иванов', '10 Петров', '11 Сидоров', '9 Морозов',  
 '9 Беляев', '10 Борисов', '11 Смирнов']  
  
make\_list\_of\_students(lst)

9 Иванов

9 Морозов

9 Беляев

10 Петров

10 Борисов

11 Сидоров

11 Смирнов

Среди всех сортировок, которые рассматриваются академически, есть сортировка Radix sort.

Приветствую всех любителей алгоритмов. Хочу Вам поведать о своих изысканиях на тему сортировок в целом и углубиться в рассмотрение radix сортировки.

Будучи разработчиком с многолетним стажем, всё чаще стал сталкиваться со странной тенденцией в разработке программного обеспечения:

Не смотря на развитие аппаратной части современных вычислителей и усовершенствовании алгоритмов, в целом производительность кода не только не выросла, но и местами изрядно деградировала.

Полагаю это связано с общей идеей отдать предпочтение быстрому программированию с использованием всё более мощных фреймворков и сверхвысокоуровневых / скриптовых языков программирования.

Языки подобные Ruby или Python невероятно удобны для разработчика. Множество 'синтаксического сахара', я бы даже сказал 'Мёда', ускоряют разработку в разы, если не на порядки, но какой ценой.

Как пользователя, меня напрягает низкая тепловая эффективность кода, про объёмы потребляемой памяти просто промолчу, однако главный ресурс человечества – время.

Оно бесследно исчезает в бесконечных абстракциях, похищается анализаторами кода, вычищается умными сборщиками мусора.

Я не призываю вернуться в прошлое, отказавшись от благ современной разработки, писать “дорогой” код, лишь предлагаю задуматься над возможным устранением “бутылочных горлышек” производительности там, где это возможно в типовых задачах.

Часто этого можно достичь, оптимизировав высоконагруженные участки кода.

Поразрядная сортировка – разбивает число на цифры.

Т.е. число 256 на 2,5 и 6.

И она упорядочивает эти элементы в соответствии со значениями этих цифр.

Сравнение вообще нет, её удобно рассматривать в 10-чной системе счисления.

В этой сортировке все элементы делятся на 10 групп (корзин).

0,1,…9

Сначала анализируется 1-я цифра в числе

Сначала идут с нулями, потом с единицами, с двойками и т.д.

Т.е некая сортировка по к примеру для трехзначных чисел по сотням

Потом снова, но уже по второй цифре, но в каждой корзине цифр.

Количество таких сортировок равно количеству цифр в числе.

Если некоторая цифра меньше, то слева дописываются незначащие нули

3 – 003

Потом всё это дело объединяется.

И это очень прикольно выглядит. Реализуем. Легко

l = [91,46,85,15, 22, 92,35,31]  
  
*# 0:  
# 1: 91 31  
# 2: 22 92  
# 3:  
# 4:  
# 5: 85 35  
# 6: 46  
# 7:  
# 8:  
# 9:  
# \_\_ 91 31 22 92 85 35 46  
  
# 0:  
# 1:  
# 2: 22  
# 3: 31 25  
# 4: 46  
# 5:  
# 6:  
# 7:  
# 8: 85  
# 9: 91 92  
  
# 22 31 25 46 85 91 92*

В лоб

from random import randint  
  
l = [randint(10, 99) for i in range(10)]  
print(l)  
l0 = l1 = l2 = l3 = l4 = l5 = l6 = l7 = l8 = l9 = []  
  
t = [[] for i in range(10)]  
  
for i in l:  
 t[i % 10].append(i)  
print(t)  
t2 = []  
  
for e in t:  
 t2 += e  
  
t3 = [[] for i in range(10)]  
for i in t2:  
 t3[i // 10].append(i)  
  
print(t3)  
t4 = []  
for e in t3:  
 t4 += e  
  
print(t4)

А теперь в ОО-парадигме с использованием очередей

from lesson\_queue import Queue  
  
  
class RadixSort:  
  
 def \_\_init\_\_(self, n):  
 self.bins = list()  
 self.nums = n  
  
 for i in range(10):  
 self.bins.append(Queue())  
  
 def distribute(self, digit):  
 for i in range(10):  
 if digit == 1:  
 self.bins[self.nums[i] % 10].enqueue(self.nums[i])  
 else:  
 self.bins[self.nums[i] // 10].enqueue(self.nums[i])  
  
 def collect(self):  
 i = 0  
 for digit in range(10):  
 while not self.bins[digit].is\_empty():  
 self.nums[i] = self.bins[digit].dequeue()  
 i += 1  
  
 def show(self):  
 return ''.join(str(self.nums))  
  
  
from random import randint  
  
l = [randint(10, 99) for i in range(10)]  
print(l)  
t = RadixSort(l)  
t.distribute(1)  
t.collect()  
print(t.show())  
  
t.distribute(10)  
t.collect()  
print(t.show())

Результат:

[99, 38, 51, 88, 70, 92, 53, 17, 26, 20]

[70, 20, 51, 92, 53, 26, 17, 38, 88, 99]

[17, 20, 26, 38, 51, 53, 70, 88, 92, 99]

## Практикум

Объединение танцоров по парам

Дан текстовый файл со списком танцоров. В каждой строке сначала записан пол танцора (М - мальчик, Д - девочка), затем – имя и фамилия танцора. Все значения разделены пробелами

Например:

Д Зина Корзинкина

М Вася Пупкин

М Федя Сумкин

М Вова Морковкин

Д Катя Петрова

М Петя Зайцев

Д Маша Простоквашина

М Дима Дрезина

М Коля Лукошкин

М Миша Косолапов

Д Наташа Блинова

Необходимо:

* создать и описать класс DanceFloor, которому в конструктор передаётся имя файла
* в классе DanceFloor создать и описать метод dance, который должен напечатать информацию о получившихся парах и о тех танцорах, которые остались без пары (их количество и имя первого танцора в очереди)
* Учтите, что:
  + мальчики танцуют только с девочками
  + девочки танцуют только с мальчиками

## Как это должно работать

cmd

>>> process = DanceFloor("dancers.txt")

>>> process.dance();

Образовались следующие пары:

Зина и Вася

Катя и Федя

Маша и Вова

Наташа и Петя

Мальчиков в очереди: 3 и первый из них — Дима

## Решение

from lesson\_queue import Queue  
  
  
class Dancer:  
 def \_\_init\_\_(self, name, sex):  
 self.name = name  
 self.sex = sex  
  
  
class Dance:  
 def \_\_init\_\_(self, filename):  
 self.male = Queue()  
 self.female = Queue()  
 self.\_\_get\_dancer(filename)  
  
 def \_\_get\_dancer(self, filename):  
 with open(filename, 'r', encoding='utf-8') as f:  
 lines = f.read().split('\n')  
 for line in lines:  
 dancer = line.strip().split(' ')  
 if dancer[0].upper() == 'Д':  
 self.female.enqueue(Dancer(dancer[1], dancer[0]))  
 else:  
 self.male.enqueue(Dancer(dancer[1], dancer[0]))  
  
 def dance(self):  
 print('Образовались следующие пары')  
 while not self.female.is\_empty() and not self.male.is\_empty():  
 s = f'{self.female.dequeue().name} и {self.male.dequeue().name}'  
 print(s)  
 if self.female.size() > 0:  
 print(f'Девочек в очереди {self.female.size()}', end='')  
 print(f'и первая из них \u2014 {self.female.front().name}')  
  
 if self.male.size() > 0:  
 print(f'Мальчиков в очереди {self.male.size()}', end=' ')  
 print(f'и первый из них \u2014 {self.male.front().name}')  
  
  
process = Dance('txt.txt')  
process.dance()

Результат:

Образовались следующие пары

Зина и Вася

Катя и Федя

Маша и Вова

Наташа и Петя

Мальчиков в очереди 3 и первый из них — Дима

Очередь с приоритетом

Пример очередь в поликлинику, в которую попали люди пришедшие по талончикам и все остальные, у которых сломались госуслуги, телефон и не работает интернет.

Мы должны всех обслужить без драки!

Включая, тех кто пришел раньше, тех у кого есть талон и вип-персон, у которых есть приоритет от главврача или от министра обороны, чвк вагнер, ну или как его зовут министра здравоохранения Краснодарского края.

Как это автоматизировать?

Два варианта.

По простоте душевной – отнаследуем очередь!

class PrirityQueue(Queue):  
 pass  
  
class Client:  
 def \_\_init\_\_(self, name, prority):  
 self.name = name  
 self.priority = prority  
  
  
q = PrirityQueue()  
q.enqueue(Client('Bob', 2))  
q.enqueue(Client('Tob', 1))  
  
process = Dance('txt.txt')  
process.dance()

Примерный подход таков.

У как производит сортировку для тех, у кого одинаковый приоритет?

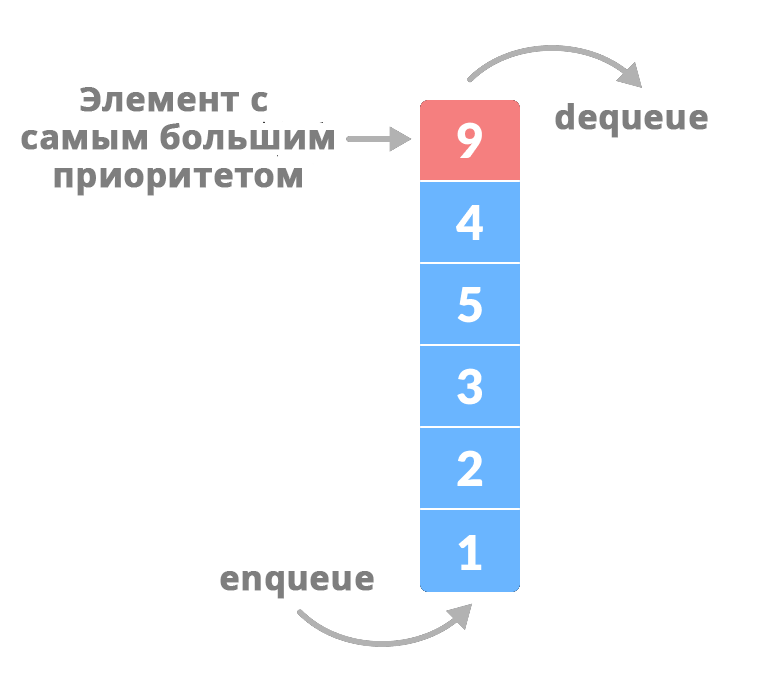
Каждый раз при добавлении – пересортировывать?

Или

Очереди с приоритетом — разновидность очередей, в которой у каждого элемента есть свой приоритет. Обслуживаются они в соответствии со своими приоритетом. Если у элементов одинаковый приоритет, то обслуживаются они по их порядку в очереди.

Значение элемента, как правило, и определяет его приоритет.

То есть, у элемента с самым большим значением самый высокий приоритет. Правда, это не всегда так. Самый высокий приоритет может быть у элемента и с самым малым значением. В остальных случаях мы можем задавать приоритеты элементам по своему усмотрению.



## Разница между очередью с приоритетом и обычной очередью

Обычная очередь подчиняется принципу FIFO «первый вошел — первый вышел». В очередях с приоритетом элементы удаляются в соответствии с их приоритетом. То есть, элемент с самым высоким приоритетом удаляется из очереди в первую очередь.

## Реализация очереди с приоритетом

Очереди с приоритетом можно реализовать с помощью следующих структур данных: массив, связный список, куча и двоичное дерево поиска. Среди всех этих структур выделяется куча — это самый эффективный способ реализации очереди с приоритетом.мы

## Дек. Бардачная очередь

Входит можно с любой стороны, выходит с любой тоже

Работа с классом collection.deque

Класс collection.deque используется для реализации двусторонней очереди, которая поддерживает добавление и удаление элемента с обоих концов. Для завершения процесса требуется O (1) времени.

Класс deque можно использовать как в очереди, так и в виде стека, поскольку он эффективно удаляет и добавляет элементы. Collection.deque может быть хорошим выбором для структуры данных в стандартной библиотеке Python.

Пример –

from collections import deque

que = deque()

que.append('Apple')

que.append('Mango')

que.append('Banana')

print(que)

deque(['Apple ', 'Mango', 'Banana'])

print(que.popleft())

print(que.popleft())

print(que.popleft())

que.popleft()

Выход:

deque(['Apple', 'Mango', 'Banana'])

Apple

Mango

Banana

Traceback (most recent call last):

File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/Queue.py", line 101, in <module>

que.popleft()

IndexError: pop from an empty deque

Класс multiprocessing.Queue

Класс multiprocessing.Queue используется для реализации элементов в очереди для параллельной обработки многопоточными рабочими процессами. Multiprocessing.Queue обменивается данными между процессами и может хранить любой объект с возможностью сериализации. Давайте разберемся в следующем примере.

Пример –

from multiprocessing import Queue

que = Queue()

que.put('Apple')

que.put('Mango')

que.put('Banana')

print(que)

print(que.get())

print(que.get())

print(que.get())

Выход:

<multiprocessing.queues.Queue object at 0x000002CA073356A0>

Apple

Mango

Banana

Приоритетная очередь в Python

Очередь с приоритетом – это особый тип очереди в структуре данных. Как следует из названия, она сортирует элементы и удаляет их в соответствии с приоритетами. В отличие от обычной, она извлекает элемент с наивысшим приоритетом вместо следующего элемента. Приоритет отдельных элементов определяется порядком их ключей.

Очереди приоритетов наиболее полезны для решения проблем планирования, когда некоторые задачи будут выполняться на основе приоритетов. Например, задача операционной системы – лучший пример очереди с приоритетом. Она выполняет высокий приоритет над задачами с более низким приоритетом (загрузка обновлений в фоновом режиме). Планировщик задач может разрешить выполнение задач с наивысшим приоритетом первыми.

Есть несколько способов реализовать приоритетную очередь в Python.

Список, отсортированный вручную

Мы можем использовать отсортированный список Python в качестве очереди приоритетов, чтобы быстро идентифицировать и удалять меньший и самый большой элемент. Но вставка нового элемента происходит медленно, так как требует O (n) операций. Следовательно, отсортированный список может быть эффективным, когда в приоритетную очередь будет сделано мало вставок.

Разберем в следующем примере –

Пример –

pri\_que = []

pri\_que.append((2, 'Apple'))

pri\_que.append((1, 'Mango'))

pri\_que.append((3, 'Banana'))

# NOTE: Remember to re-sort every time

# a new element is inserted.

pri\_que.sort(reverse=True)

while pri\_que:

next\_item = pri\_que.pop()

print(next\_item)

Выход:

(1, 'Mango')

(2, 'Apple')

(3, 'Banana')

Класс queue.PriorityQueue

Эта приоритетная очередь реализует внутреннее использование heapq и разделяет те же временные и пространственные сложности.

Разница в том, что приоритетная очередь скоординирована и обеспечивает семантику блокировки для поддержки нескольких одновременных событий и потребителей.

Пример –

from queue import PriorityQueue

q = PriorityQueue()

q.put((2, 'Apple'))

q.put((1, 'Banana'))

q.put((3, 'Mango'))

while not q.empty():

next\_item = q.get()

print(next\_item)

Выход:

(1, 'Banana')

(2, 'Apple')

(3, 'Mango')

Мы можем выбрать любую реализацию приоритетной очереди в программе Python, но имейте в виду, что queue.PriorityQueue – хороший выбор по умолчанию.

Заключение

Мы обсудили все основные концепции очереди и ее реализации. Она похожа на стандартный список, но по производительности всегда лучше. Мы также определили приоритетную очередь и различные способы ее реализации.

Источник: https://pythonpip.ru/examples/ochered-python

## Библиотечные модули collections и collections.abc

Модуль collections содержит специализированный контейнер типов данных, который может быть использован для замены контейнеров общего назначения Python (dict, tuple, list, и set). Мы изучим следующие части этого замечательного модуля:

Counter

ChainMap

defaultdict

deque

namedtuple

OrderedDict

Также существует наследованный модуль коллекций под названием abc, или Abstract Base Classes. Мы рассмотрим чуть позже раз. Давайте начнем с контейнера ChainMap!

## Counter

Одна из распространённых задач, для которой начинающие питонисты придумывают собственные решения, – подсчёт элементов последовательности: списка, строки символов и т. д.

Если нужно что-то посчитать, определить количество вхождений или наиболее (наименее) часто встречающихся элементов, используйте объекты класса Counter. Создаются они с помощью конструктора collections.Counter().

Функция принимает итерируемый аргумент и возвращает словарь, в котором ключами служат индивидуальные элементы, а значениями – количества повторений элемента в переданной последовательности. Посчитаем, сколько раз встречается каждая буква в слове «абракадабра»:

list\_of\_letters = list('абракадабра')

letter\_cnt = collections.Counter(list\_of\_letters)

letter\_cnt

Counter({'а': 5, 'б': 2, 'р': 2, 'к': 1, 'д': 1})

Обращение к ключам происходит аналогично обычному словарю:

letter\_cnt['а']

5

Если элемент отсутствовал в последовательности, при обращении по ключу счётчик не вызовет исключение, а вернет нулевое значение:

letter\_cnt['ю']

0

Присвоение нуля ключу не удаляет это значение, а создаёт соответствующую пару:

letter\_cnt['в'] = 0

letter\_cnt

Counter({'а': 5, 'б': 2, 'р': 2, 'к': 1, 'д': 1, 'в': 0})

Чтобы удалить пару key-value, используем del:

del letter\_cnt['в']

letter\_cnt

Counter({'а': 5, 'б': 2, 'р': 2, 'к': 1, 'д': 1})

В качестве аргумента конструктор принимает не только последовательность, но и словарь, содержащий результаты подсчёта:

emotion\_cnt = collections.Counter({'like':2, 'dislike':3})

emotion\_cnt

Counter({'like': 2, 'dislike': 3})

Метод elements() преобразует результаты подсчета в итератор:

list(emotion\_cnt.elements())

['like', 'like', 'dislike', 'dislike', 'dislike']

Метод most\_common(n) ищет n самых повторяющихся элементов. Найдём для примера три наиболее частых символа:

# без передачи аргумента выводятся все элементы

# в порядке от наиболее частых к наиболее редким

letter\_cnt.most\_common(3)

[('а', 5), ('б', 2), ('р', 2)]

Метод возвращает список кортежей вида (ключ, число повторений).

Нередко интерес представляют не самые частотные, а уникальные значения, самые редкие элементы. Их можно найти срезом с шагом -1:

letter\_cnt.most\_common()[:-3:-1]

[('д', 1), ('к', 1)]

Счётчики складываются и вычитаются друг из друга:

letter\_cnt + emotion\_cnt

Counter({'а': 5, 'dislike': 3, 'б': 2, 'р': 2, 'like': 2, 'к': 1, 'д': 1})

emotion\_cnt - collections.Counter(like=1, dislike=3)

Counter({'like': 1})

Операнд & даст минимальные значения для одних и тех же подсчитываемых элементов, операнд | – максимальные:

c = collections.Counter(a=4, b=2, c=0, d=-2)

d = collections.Counter(a=1, b=2, c=3, d=4)

c & d

Counter({'b': 2, 'a': 1})

c | d

Counter({'a': 4, 'd': 4, 'c': 3, 'b': 2})

Как видно из примера, счётчику можно передавать отрицательные значения. Однако для перечисленных операций хранятся только положительные подсчеты. Нулевые или отрицательные значения обычно приходится хранить при вычитании, что реализовано в методе subtract():

c.subtract(d)

c

Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})

Обратите внимание, что метод subtract() обновляет сам счётчик, а не создает новый.

Распространненные шаблоны применения Counter:

sum(letter\_cnt.values()) # число всех посчитанных элементов

11

list(letter\_cnt) # список уникальных элементов исходной последовательности

['а', 'б', 'р', 'к', 'д']

set(letter\_cnt)

{'а', 'б', 'д', 'к', 'р'}

dict(letter\_cnt) # счетчик это подкласс словаря, можно преобразовать в обычный dict

{'а': 5, 'б': 2, 'р': 2, 'к': 1, 'д': 1}

Унарные операции оставляют только положительные или отрицательные подcчёты:

+c # способ вывести положительные подсчеты

Counter({'a': 3})

-c # способ вывести отрицательные подсчеты

Counter({'c': 3, 'd': 6})

c.clear() # Очищаем счетчик

c

Counter()

## Словарь со значением по умолчанию (defaultdict)

Что будет, если обратиться к словарю по ключу, которого в нем ещё нет?

Верно, исключение KeyError:

d = dict()

d['name'] = 'James'

d['surname'] = 'Bond'

d['patronymic']

KeyError Traceback (most recent call last) <...>

Если нет нужды отлавливать исключение, достаточно использовать альтернативный вариант словаря – collections.defaultdict.

Соответствующему конструктору в качестве аргумента передается тип элемента по умолчанию:

d = collections.defaultdict(str)

d['name'] = 'James'

d['surname'] = 'Bond'

d['patronymic']

''

d

defaultdict(str, {'name': 'James', 'surname': 'Bond', 'patronymic': ''})

Таким образом, для ключей, к которым происходит обращение, конструктор поставит в соответствие дефолтный элемент данного типа. В случае str – пустая строка, для целых чисел – 0 и т. д.

Обычные словари имеют метод setdefault(), который позволяет добиться того же результата, но его использование делает программный код менее наглядным и замедляет исполнение.

Помимо str и int, defaultdict часто используют в связке с пустым списком, чтобы начинать добавление элементов без лишнего кода:

dict\_of\_lists = collections.defaultdict(list)

for i in range(5):

... dict\_of\_lists[i].append(i)

...

dict\_of\_lists

defaultdict(<class 'list'>, {0: [0], 1: [1], 2: [2], 3: [3], 4: [4]})

Можно видеть, что при таком подходе нет необходимости ни проверять наличие соответствующих ключей, ни создавать предварительно пустые списки.

Словарь с памятью порядка добавления элементов (OrderedDict)

Ощутимость пользы OrderedDict так повлияла на обычный dict, что в новых версиях Python различий между ними становится всё меньше. В былые времена OrderedDict кардинально отличался от обычного словаря тем, что умел запоминать порядок вставки. Но с версии Python 3.6 на [это способен и обычный словарь](https://docs.python.org/3.6/whatsnew/3.6.html#new-dict-implementation). Однако некоторые различия между ними все равно остаются:

1. Обычный dict был разработан, чтобы быть лучшим в операциях, связанных с [мапированием](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%BF%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \t "_blank). Отслеживание порядка вставки для него – дело вторичное. И наоборот, OrderedDict хорош в операциях переупорядочения, а эффективность, скорость итераций и производительность не главное.
2. Алгоритмически OrderedDict может обрабатывать частые операции переупорядочения лучше, чем dict.

Так как OrderedDict это упорядоченная последовательность, объекты содержат соответствующие методы, реорганизующие структуру:

1. popitem(last=True) – удаляет последний элемент если last=True, и первый, если last=False
2. move\_to\_end(key, last=True) – переносит ключ key в конец, если last=True, и в начало, если last=False

d = collections.OrderedDict.fromkeys('abcde')

d.move\_to\_end('b')

''.join(d.keys())

'acdeb'

d.move\_to\_end('b', last=False)

''.join(d.keys())

'bacde'

Контейнер словарей (ChainMap)

ChainMap – это класс, который дает возможность объединить несколько сопоставлений вместе таким образом, чтобы они стали единым целым. Если вы обратитесь к документации, то увидите, что данный класс принимает \*\*maps\*.

Это значит, что ChainMap будет принимать любое количество сопоставлений или словарей и превращать их в единое обновляемое представление. Давайте взглянем на пример, чтобы вы могли увидеть, как это работает:

from collections import ChainMap  
  
car\_parts = {  
 'hood': 500,  
 'engine': 5000,  
 'front\_door': 750  
}  
  
print(car\_parts)  
  
car\_options = {  
 'A/C': 1000,  
 'Turbo': 2500,  
 'rollbar': 300  
}  
  
car\_accessories = {  
 'cover': 100,  
 'hood\_ornament': 150,  
 'seat\_cover': 99  
}  
  
car\_pricing = ChainMap(car\_accessories, car\_options, car\_parts)  
  
print(car\_pricing)  
print(car\_pricing['hood']) *# 500*

Здесь мы импортировали ChainMap из модуля collections. Затем 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. Это весьма полезно в тех случаях, когда вам нужно установить настройки по умолчанию.

Давайте представим, что нам нужно создать приложение, которое имеет определенные настройки по умолчанию. Приложение также будет знать о переменных среды операционной системы.

Если существует переменная среды, которая соответствует значению ключа, который расположен в нашем приложении по умолчанию, тогда среда переопределит наши настройки по умолчанию.

Теперь давайте представим, что мы можем передавать аргументы нашему приложению. Эти аргументы имеют преимущество над средой и настройками по умолчанию. Это тот случай, когда ChainMap представлен во всей красе.

После разговора о словарях самое время обсудить класс, умеющий объединять словари в надструктуру – ChainMap. При этом получается не один общий словарь, а их совокупность, в которой каждый словарь остаётся независимой составляющей:

letters = {'a':1, 'b':2}

vowels = {'a':1, 'b':0, 'c':0, 'd': 0, 'e':1}

chain = collections.ChainMap(letters, vowels)

chain

ChainMap({'a': 1, 'b': 2}, {'a': 1, 'b': 0, 'c': 0, 'd': 0, 'e': 1})

При обращении к ChainMap по ключу одного из словарей, происходит поиск значения среди всех словарей, при этом нет необходимости указывать конкретный словарь:

chain['e']

1

При поиске ChainMap выводит первое найденное значение (проходя словари по очереди добавления). В том числе если в словарях несколько одинаковых ключей:

chain['b']

2

Изменение содержания словаря изменяет и ChainMap. Нет необходимости перезаписывать надструктуру:

letters['c'] = 3

chain

ChainMap({'a': 1, 'b': 2, 'c': 3}, {'a': 1, 'b': 0, 'c': 0, 'd': 0, 'e': 1})

Так как ChainMap это комбинация словарей, логично, что у неё есть методы keys() и values():

list(chain.keys())

['c', 'd', 'a', 'e', 'b']

list(chain.values())

[3, 0, 1, 1, 2]

Значения values соответствуют списку keys, как это было описано выше. То есть в случае несколько совпадающих ключей, выводится значение для первого из словарей, где встречается этот ключ.

При необходимости расширить составленный ранее ChainMap можно методом new\_child():

consons = {'a':0, 'b':1, 'c':1}

chain.new\_child(consons)

ChainMap({'a': 0, 'b': 1, 'c': 1}, {'a': 1, 'b': 2, 'c': 3}, {'a': 1, 'b': 0, 'c': 0, 'd': 0, 'e': 1})

Обратите внимание, что метод не обновляет старую структуру, а создаёт новую.

Counter

Модуль collections также предоставляет нам небольшой аккуратный инструмент, который поддерживает быстрый и удобный в пользовании калькулятор. Этот инструмент называется Counter. Вы можете запустить его против большинства итерируемых. Давайте попробуем взглянуть на него в строке.

Python

from collections import Counter

a = Counter('superfluous')

# Counter({'u': 3, 's': 2, 'e': 1, 'l': 1, 'f': 1, 'o': 1, 'r': 1, 'p': 1})

print(a)

counter = Counter('superfluous')

print(counter['u']) # 3

В данном примере мы импортировали Counter из модуля collections и передали его строке. Это возвращает нам объект Counter, который является наследуемым классом словаря Python. Когда мы запускаем эту же команду, но назначаем её счетчик переменной, чтобы доступ к словарю был несколько проще. В данном случае, мы видим, что буква “u” встречается три раза в нашем примере. Класс Counter предоставляет несколько методов, которые могут вас заинтересовать. Например, вы можете вызывать элементы, которые будут выполнять итерацию над элементами, расположенными в словаре, но в произвольном порядке. Вы можете подумать, что эта функция является своего рода скремблером, так как выдача в этом случае представлена как скремблированная версия строки.

Python

print(list(counter.elements()))

# ['e', 'l', 'f', 'o', 'r', 's', 's', 'p', 'u', 'u', 'u']

Еще один полезный метод это most\_common. Вы можете спросить Counter о том, какие объекты являются наиболее распространенными, передав число, отображающее наиболее часто повторяющие объекты “n”:

Python

print(counter.most\_common(2))

# [('u', 3), ('s', 2)]

Здесь мы попросили наш Counter выяснить, какие два объекта являются наиболее повторяемыми. Как вы видите, мы получили список кортежей, в котором указано, что “u” встречается 3 раза, а “s” – два раза. Еще один метод, который я хотел бы рассмотреть, это метод subtract.

# Двусторонняя очередь (deque)

Объект типа deque (читается как «дэк», [двусторонняя или двусвязная очередь](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D1%83%D1%85%D1%81%D1%82%D0%BE%D1%80%D0%BE%D0%BD%D0%BD%D1%8F%D1%8F_%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C)) является усовершенствованным вариантом списка с оптимизированной вставкой/удалением элементов с обоих концов. Реализация deque оптимизирована так, что операции слева и справа имеют примерно одинаковую производительность O(1). Добавление новых элементов в конец происходит не сильно медленнее, чем во встроенных списках, но добавление в начало выполняется существенно быстрее.

Наш deque поддерживает аргумент maxlen, который устанавливает границы для deque. В противном случае deque будет расти до произвольных размеров. Когда ограниченный deque заполнен, любые новые объекты, которые были добавлены, вызовут такое же количество элементов, которые выскочат с другого конца. Вот основное правило: если вам нужно что-то быстро дописать или вытащить, используйте deque. Если вам нужен быстрый случайный доступ, используйте list. Давайте уделим пару минут, и посмотрим, как мы можем создавать и использовать deque.

from collections import deque

import string

d = deque(string.ascii\_lowercase)

for letter in d:

print(letter)

Здесь мы импортируем deque из нашего модуля collections, а также модуль string. Для того, чтобы создать экземпляр deque, нам нужно передать его итерируемой. В данном случае, мы передали его string.ascii\_lowercase, и получили список всех строчных букв в алфавите. Наконец, мы сделали цикл над deque и вывели каждый объект. Теперь давайте взглянем на несколько методов, которыми обладает deque.

seq = list("bcd")

deq = collections.deque(seq)

deq

deque(['b', 'c', 'd'])

deq.append('e') # добавление в конец

deq.appendleft('a') # добавление в начало (левый конец)

deq

deque(['a', 'b', 'c', 'd', 'e'])

Чтобы добавлять не одиночный элемент, а группу итерируемого объекта iterable используйте соответственно extend(iterable) и extendleft(iterable).

Аналогично методу append() метод pop() для deque работает с обоих концов:

deq.pop()

deq.popleft()

deq

deque(['b', 'c', 'd'])

Если нужно посчитать число вхождений элемента в последовательность, применяем метод count():

deq.count('b'), deq.count('a')

(1, 0)

Кроме перечисленных, доступны следующие методы:

1. remove(value) – удаление первого вхождения value
2. reverse() – разворачивает очередь)
3. rotate(n=1) – последовательно переносит n элементов из начала в конец (если n отрицательно, то с конца в начало). В этом поведение deque напоминает [кольцевой связный список](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA#%D0%9A%D0%BE%D0%BB%D1%8C%D1%86%D0%B5%D0%B2%D0%BE%D0%B9_%D1%81%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA)

Очередь deque имеет аргумент maxlen, позволяющий ограничить ее размер. При заполнении ограниченной очереди добавление n новых объектов «слева» вызовет удаление n элементов справа.

Ограниченные очереди обеспечивают функциональность, похожую на tail-фильтр в Unix:

def tail(filename, n=10):

"""Возвращает n последних строк файла'"""

with open(filename) as f:

return collections.deque(f, n)

Давайте закончим этот раздел и взглянем на пример, основанный на документации Python:

Python

from collections import deque

def get\_last(filename, n=5):

"""

Возвращаем последние N кол-во строк из файла

"""

try:

with open(filename) as f:

return deque(f, n)

except OSError:

print("Файл не открывается: {}".format(filename))

raise

Этот код работает по схожему принципу с программой-хвостом Linux. Здесь мы передаем имя файла нашему скрипту вместе с n количеством строк, которые мы хотим вернуть. Наш deque ограничен той или иной цифрой, которую мы указываем как n. Это значит, что как только deque заполнится, когда новые строки прочитаны и добавлены в deque, старые строки выпадают из другого конца и отбрасываются. Я также завернул открываемый в операторе файл в простой обработчик исключений, так как очень легко выполнить передачу по неверному пути. Таким образом, мы поймаем несуществующие файлы, к примеру. Теперь мы готовы идти дальше и приступить к изучению namedtuple.

Другой шаблон применения deque – хранение последних добавленных элементов с выбрасыванием более старых. Пример компактной и быстрой реализации функции [скользящего среднего](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D0%BE%D0%BB%D1%8C%D0%B7%D1%8F%D1%89%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%BD%D1%8F%D1%8F):

import itertools

def moving\_average(iterable, n=3):

# moving\_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0

it = iter(iterable)

d = collections.deque(itertools.islice(it, n-1))

d.appendleft(0)

s = sum(d)

for elem in it:

s += elem - d.popleft()

d.append(elem)

yield s / n

Алгоритм распределения нагрузки [Round-robin](https://ru.wikipedia.org/wiki/Round-robin_(%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC)" \t "_blank) можно реализовать с помощью итераторов, хранящихся в deque. Значения выводятся из активного итератора в нулевой позиции. Если этот итератор исчерпан, его можно удалить методом popleft (); в противном случае его можно циклически «провернуть» до конца методом rotate():

def roundrobin(\*iterables):

"roundrobin('ABC', 'D', 'EF') --> A D E B F C"

iterators = collections.deque(map(iter, iterables))

while iterators:

try:

while True:

yield next(iterators[0])

iterators.rotate(-1)

except StopIteration:

# Удалить "закончившийся" итератор

iterators.popleft()

## Именованный кортеж и функция namedtuple()

сфокусируемся на namedtuple, который можно использовать для замены кортежей Python. Разумеется, namedtuple действует не так примитивно, как может показаться на первый взгляд. Я видел некоторых программистов, которые используют его как struct. Если вы не работали с языком, в котором есть struct, тогда нужно объяснить немного подробнее. Когда мы говорим о struct, мы имеем ввиду сложный тип данных, который группирует список переменных под одним именем. Давайте взглянем на пример того, как создается namedtuple, чтобы понять принцип его работы:

namedtuple() – функция-фабрика для создания именованных кортежей. Этот тип данных похож на struct в других языках программирования:

cols = ['fname', 'pname', 'lname', 'age']

User = collections.namedtuple('User', cols)

user1 = User('Петр', 'Иванович', 'Сидоров', 30)

user1

User(fname='Петр', pname='Иванович', lname='Сидоров', age=30)

user1.lname

Сидоров

Point = collections.namedtuple('Point', ['x', 'y'])

p = Point(3, 4)

p.x\*\*2 + p.y\*\*2

25

Именованные кортежи делают код яснее – вместо индексирования составляющие объекта вызываются по явным именам. Остаётся доступной и численная индексация:

p[0]\*\*2 + p[1]\*\*2

25

Давайте взглянем на пример того, как создается namedtuple, чтобы понять принцип его работы:

Python

from collections import namedtuple

Parts = namedtuple('Parts', 'id\_num desc cost amount')

auto\_parts = Parts(

id\_num='1234',

desc='Ford Engine',

cost=1200.00,

amount=10

)

print(auto\_parts.id\_num)

Здесь мы импортировали namedtuple из модуля collections. Далее мы вызываем namedtuple, который вернет новый наследуемый класс кортежа, но с названными полями. Так что фактически мы создали новый класс кортежа. Обратите внимание на то, что у нас имеется странная строка в качестве второго аргумента. Это список свойств, который мы хотим создать, разделенный пробелами. Теперь, когда у нас появился чудесный новый класс, давайте создадим его экземпляр! Как вы видите выше, это делается так же, как когда мы создаем объект auto\_parts. Теперь мы можем дать auto\_parts доступ к различным предметам, используя точечную нотацию, так как теперь они являются свойствами нашего класса Parts. Одно из преимуществ использования namedtuple над обычным кортежем заключается в том, что вам не нужно отслеживать индекс каждого объекта, так как каждый объект назван и доступен через свойство класса. Давайте взглянем на разницу на примере кода:

Python

auto\_parts = ('1234', 'Ford Engine', 1200.00, 10)

print(auto\_parts[2]) # выводим цену: 1200.0

id\_num, desc, cost, amount = auto\_parts

print(id\_num) # '1234'

В данном коде мы создали обычный кортеж и передали доступ к цене на двигатель автомобиля, указав Python необходимый нам индекс. Как альтернативный способ, мы можем также извлечь все что есть из кортежа используя множественное назначение. Лично я предпочитаю в таких случаях использовать namedtuple, так как это проще для мозга, и вы можете использовать метод dir() для проверки кортежа и нахождения его свойств. Попробуйте сами, и посмотрите, что у вас выйдет. Как-то раз я искал способ конвертировать словарь Python в объект и наткнулся на код, который делает что-то вроде этого:

Python

from collections import namedtuple

Parts = {

'id\_num':'1234',

'desc':'Ford Engine',

'cost':1200.00,

'amount':10

}

parts = namedtuple('Parts', Parts.keys())(\*\*Parts)

print(parts)

# Parts(amount=10, cost=1200.0, id\_num='1234', desc='Ford Engine')

Это немного странный код, так что давайте разберем его по кусочкам. В первой строке мы импортируем namedtuple, как и раньше. Далее, мы создаем словарь Parts. Пока все логично. Теперь переходим к странной части. Здесь мы создаем наш класс namedtuple и называем его ‘Parts’. Второй аргумент – это список ключей из нашего словаря. Последняя часть этой странной части кода: (Parts)\*\*. Двойная звездочка означает, что мы вызываем наш класс при помощи ключевых аргументов, в нашем случае это словарь. Мы можем разделить эту строку на две части, для большей гигиеничности:

Python

parts = namedtuple('Parts', Parts.keys())

print(parts)

# <class '\_\_main\_\_.Parts'>

auto\_parts = parts(\*\*Parts)

print(auto\_parts)

# Parts(amount=10, cost=1200.0, id\_num='1234', desc='Ford Engine')

Здесь мы делаем то же, что и раньше, за исключением того, что сначала мы создаем класс, затем мы вызываем этот класс при помощи словаря для создания объекта. Еще один момент, который я хотел бы прояснить: namedtuple также принимает аргументы verbose и rename.

Аргумент verbose – это флаг, который выводит определение класса непосредственно перед его построением, если вы укажете его значение как True.

Аргумент rename весьма полезен в тех случаях, когда вы создаете namedtuple из базы данных, или какой-нибудь другой системы, которую ваша программа не контролирует для автоматического переименования свойств для вас.

Именованные кортежи часто используются для назначения имён полей кортежам, возвращаемым модулями csv или sqlite3:

EmployeeRecord = collections.namedtuple('EmployeeRecord', 'name, age, title, department, paygrade')

import csv

for emp in map(EmployeeRecord.\_make, csv.reader(open("employees.csv", "rb"))):

print(emp.name, emp.title)

import sqlite3

conn = sqlite3.connect('/companydata')

cursor = conn.cursor()

cursor.execute('SELECT name, age, title, department, paygrade FROM employees')

for emp in map(EmployeeRecord.\_make, cursor.fetchall()):

print(emp.name, emp.title)

Структура namedtuple похожа на словарь. Посредством метода \_asdict можно представить те же данные в виде OrderedDict:

p.\_asdict()

OrderedDict([('x', 3), ('y', 4)])

Чтобы вызвать значение через строковый ключ, необязательно преобразовывать namedtuple – подходит стандартная функция getattr():

getattr(p, 'x')

3

Чтобы преобразовать словарь в именованный кортеж заданного типа, достаточно распаковать его оператором \*\*:

d = {'x': 0, 'y': 1}

Point(\*\*d)

Point(x=0, y=1)

Имена полей namedtuple перечислены в \_fields:

user1.\_fields, p.\_fields

(('fname', 'pname', 'lname', 'age'), ('x', 'y'))

С версии 3.7 можно присвоить полям [значения по умолчанию](https://docs.python.org/3/library/collections.html#collections.somenamedtuple._field_defaults).

Поскольку именованный кортеж является обычным классом Python, в него легко привнести новую функциональность или изменить старую. Например, добавим к Point расчёт гипотенузы и формат вывода данных:

class Point(collections.namedtuple('Point', ['x', 'y'])):

\_\_slots\_\_ = () # предотвращает создание словарей экземпляров

@property

def hypot(self):

return (self.x\*\*2 + self.y\*\*2) \*\* 0.5

def \_\_str\_\_(self):

return 'Point: x=%6.3f y=%6.3f hypot=%6.3f' % (self.x, self.y, self.hypot)

for p in Point(3, 4), Point(14, 5/7):

print(p)

Point: x= 3.000 y= 4.000 hypot= 5.000

Point: x=14.000 y= 0.714 hypot=14.018

Если вам пришлась по душе компактность namedtuple в сравнении с обычными классами и ваш проект может работать с версиями Python не меньше 3.7, присмотритесь к модулю [dataclasses](https://docs.python.org/3/library/dataclasses.html" \t "_blank). Эта встроенная библиотека предоставляет декоратор и функции для автоматического добавления в пользовательские классы сгенерированных специальных методов, таких как \_\_init\_\_() и \_\_repr\_\_().

Резюме

Подведём итог нашему рассказу об основных составляющих модуля collections:

1. [Counter](https://docs.python.org/3/library/collections.html#counter-objects) – инструмент подсчёта неизменяемых объектов. Используйте, если нужно определить количество вхождений или число наиболее (наименее) часто встречающихся элементов.
2. [defaultdict](https://docs.python.org/3/library/collections.html#defaultdict-objects) – словарь, умеющий при вызове отсутствующего ключа вместо вызова исключения KeyError записывать значение по умолчанию (работает быстрее, чем метод setdefault()).
3. [OrderedDict](https://docs.python.org/3/library/collections.html#ordereddict-objects) – словарь с памятью порядка добавления элементов, умеющий переупорядочивать элементы лучше, чем dict.
4. [ChainMap](https://docs.python.org/3/library/collections.html#collections.ChainMap) – контейнер комбинаций словарей с поиском, обобщением ключей и элементов.
5. [namedtuple()](https://docs.python.org/3/library/collections.html#namedtuple-factory-function-for-tuples-with-named-fields) – функция-фабрика для создания именованного кортежа. Это один из простейших способов сделать код более ясным: использовать вместо индексов имена.
6. [deque](https://docs.python.org/3/library/collections.html#deque-objects) – двусторонняя очередь – список, оптимизированный для вставки и удаления элементов с обоих концов с методом подсчёта вхождений
7. [UserDict](https://docs.python.org/3/library/collections.html#userdict-objects), [UserList](https://docs.python.org/3/library/collections.html" \l "userlist-objects" \t "_blank), [UserString](https://docs.python.org/3/library/collections.html" \l "userstring-objects" \t "_blank) – не заслуживающие развёрнутого описания обертки над стандартными объектами словарей, списков и строк для беспроблемного наследования (прямое наследование встроенным типам dict, list, str чревато ошибками, связанными с игнорированием переопределения методов).

Также у модуля collections имеется наследованный модуль коллекции абстрактных базовых классов сollections.abc. Но это тема отдельного разговора.

## Модуль collections.abc

Набор абстрактных классов для контейнеров. Их можно использовать для создания интерфейсов, но, в основном, ими пользуются как Type hints.

Этот модуль предоставляет абстрактные базовые классы , которые можно использовать для проверки того, предоставляет ли класс конкретный интерфейс; например, является ли он хешируемым или является ли это сопоставлением.  
  
Новый класс не обязан наследоваться от Sequence, ему достаточно реализовать те же методы, что есть в Sequence, issubclass(C, Sequence) вернёт True.

