Python: Документация

Барсуков Дмитрий

Февраль 2026

Содержание

1 Архитектура конспекта **2**

2 Структуры данных **3**

2.1 Хеш-таблица 3

2.2 Множество (set) 6

Архитектура конспекта

Этот конспект — не пересказ документации и не сухая теория. Это структурированная система, которая помогает не просто помнить синтаксис, а осознанно выбирать решения в алгоритмических задачах.

Главная цель — понимать, как работает инструмент, когда его применять и какие компромиссы он несёт.

Шаблон раздела структуры данных

Каждая тема описывается по единому плану, чтобы сохранялась логика анализа, было проще сравнивать разные структуры и приёмы и быстро находить нужную информацию в любой теме

0. **Связь с другими структурами (если есть):** Где эта структура встречается, частью чего является или что лежит в её основе. Помогает видеть общую картину языка и выстраивать связи между темами
1. **Идея:** Что это такое? Краткое определение и главная цель конструкции. Зачем она существует и какую проблему решает
2. **Интуитивная модель:** Как это работает «под капотом»? Объяснение принципа работы без излишней технической детализации
3. **Основные операции и методы:** Перечень того, что можно делать с этой структурой: добавление, удаление, поиск, изменение и характерные для неё методы. Без оценки сложности
4. **Синтаксис:** Формальная запись и базовые примеры использования. Минимальный набор конструкций, необходимый для практики
5. **Сложность (Big O):** Оценка времени выполнения и памяти для ключевых операций. Важно для понимания эффективности кода и выбора подходящей структуры под задачу
6. **Применимость и ограничения:** Когда эта конструкция лучший выбор, когда от неё лучше отказаться, и какие компромиссы она несёт (скорость/память/читаемость)

Структуры данных

Хеш-таблица

0. Связь с другими структурами

Хеш-таблица лежит в основе структур данных:

- Множество (set)
- Словарь (dict)
- `Counter`, `defaultdict` и других производных структур

1. Идея

Хеш-таблица — это структура данных, обеспечивающая быстрый доступ к элементам по ключу, которая хранит элементы так, чтобы операции поиска/добавления/удаления обычно выполнялись за O(1). Основной принцип работы: объект → вычисление хеша → определение индекса в таблице → обращение к ячейке.

2. Интуитивная модель

Хеш-таблицу можно представить как большой массив ячеек, часть из которых остаётся пустой

Добавление элемент `x`:

1. От элемента вычисляется хеш:

```python
hash(x)
```

2. От хеша берётся остаток от деления на размер массива — получается индекс ячейки:

```python
index = hash(x) % size
```

3. Проверка:
   - Если ячейка пустая → объект записывается туда
   - Если занята другим объектом (коллизия) → применяется поиск следующей подходящей позиции
   - Если найден тот же объект → добавление не происходит

Почему так быстро?
Потому что мы “прыгаем” сразу к нужной области, а не перебираем всё подряд, как в списке

Почему хеш-таблица содержит много пустых ячеек?
Чтобы поиск следующей позиции обычно был коротким. Это и есть компромисс память ↔ скорость

Load factor (заполненность)

$$\text{Load factor} = \frac{\text{число элементов}}{\text{размер таблицы}}$$

Чем выше заполненность, тем больше коллизий, тем длиннее поиск и тем ниже эффективность

Resize (расширение таблицы)
Когда таблица становится слишком заполненной, происходит resize:

1. Создаётся таблица большего размера
2. Элементы перераскладываются по новым индексам (потому что индекс зависит от размера)
3. После этого load factor снова падает, и операции снова быстрые

Размер таблицы обычно выбирается как степень двойки: 8 → 16 → 32 → 64 → …
Это упрощает вычисление индекса и управление заполненностью

Сжатие таблицы
При массовом удалении элементов таблица может не уменьшаться автоматически
Если требуется освободить память, можно создать новую структуру на основе текущей:

```python
d = dict(d)
s = set(s)
```

Это приводит к пересборке таблицы под текущее количество элементов

3. Основные операции и методы

Хеш-таблица поддерживает следующие пользовательские операции:

1. Вставка элемента
2. Поиск элемента
3. Удаление элемента

Внутренний механизм:

4. Расширение таблицы при переполнении

В Python эти операции реализованы внутри `set` и `dict`

4. Синтаксис

В Python хеш-таблица не является самостоятельной структурой, доступной пользователю напрямую

5. Сложность (Big O)

Для корректных хешируемых ключей и нормального распределения:

1. Поиск / вставка / удаление: в среднем O(1)
2. Построение из n элементов: O(n)
3. Resize: O(n), но происходит редко, поэтому средняя сложность вставки остаётся O(1)

Память: O(n), с дополнительным запасом пустых ячеек + служебные данные
Важно: В худшем случае (при большом числе коллизий) операции могут деградировать до O(n)

6. Применимость и ограничения

Когда это лучший выбор

1. Быстрая проверка наличия элемента
2. Нужен быстрый доступ по ключу
3. Нужны частоты / группировки
4. Хранение уникальных ключей

Ограничения

1. Ключи должны быть хешируемыми: числа, строки, кортежи из hashable-элементов
2. Порядок (как “отсортированный”) хеш-таблица не даёт
3. Возможна деградация до O(n) в худшем случае (редко в нормальной практике)

Компромиссы (скорость/память/читаемость)

1. Скорость отличная → память выше, чем у списка
2. Код обычно читаемый (особенно для “подсчётов/группировок”), но иногда требуется аккуратность с ключами и равенствами

Множество (set)

Идея

Множество — это структура данных, которая:

1. Хранит только уникальные элементы
2. Обеспечивает быструю проверку наличия элемента
3. Основана на хеш-таблице

Главные преимущества:

1. Нет дубликатов
2. `x in s` работает очень быстро
3. Удобные операции над множествами: пересечение, объединение, разность

Интуитивная модель

Можно представить множество как массив ячеек, большинство из которых изначально пустые. Когда мы добавляем элемент `x`:

1. От элемента вычисляется хеш:

   ```python
   hash(x)
   ```

2. От хеша берётся остаток от деления на размер массива — получается индекс ячейки:

   ```python
   index = hash(x) % size
   ```

3. Проверка:
   - Если ячейка пустая → элемент записывается туда
   - Если занята другим элементом (коллизия) → применяется probing (поиск другой позиции)
   - Если найден тот же элемент → добавление не происходит

По такому же принципу работает:

- Поиске элемента `x in s`
- Удаление элемента `s.remove(x)`

Изначально размер таблицы небольшой, но по мере заполнения множество увеличивается.
Обычно размер — степени двойки: 8 → 16 → 32 → 64 → ...

Когда происходит resize:

1. Создаётся новая таблица большего размера, элементы перекладываются в неё
2. Их хеш не пересчитывается, но пересчитывается их индекс (потому что изменился размер таблицы)

Сложность resize — O(n)
Но так как resize происходит редко, операция добавления остаётся: амортизированно O(1)

Если мы добавили 10 миллионов элементов, потом удалили почти всё. Таблица обычно не сжимается обратно автоматически. Если нужно реально уменьшить память, можно пересобрать множество:

```python
s = set(s)
```

Это создаст новую таблицу под текущий размер. Сложность — O(n), где n — текущее число элементов.

Синтаксис

```python
nums_set = {1, 2, 3}                          # {1, 2, 3} мн-во из чисел
symbols_set = {'a', 'b', 'c', 'd'}            # {'b', 'c', 'a', 'd'} мн-во из символов
empty_set = set()                             # пустое множество (не {}, это словарь)
hello_set = set('hello')                      # hello_set = {'h', 'o', 'l', 'e'}
a = {i ** 2 for i in range(10)}               # a = {0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
from_list = set([1, 2, 2, 3])                 # {1, 2, 3} — дубли убраны
words_set = set(['hi', 'dad', 'hi', 'mum'])   # {'hi', 'dad', 'mum'}
```

Сложность (Big O)

---

1. Проверка принадлежности: `x in s` — O(1)
2. Добавление элемента: `s.add(x)` — O(1)
3. Удаление элемента: `s.remove(x)` — O(1)
   Выдаст ошибку, если x нет

3' Удаление элемента: `s.discard(x)` — O(1)
Ничего не делает, если x нет

4. Построение из списка длины n: `set(nums)` — O(n)
   Мы проходим по каждому элементу и вставляем его.

5. Получение размера множества: `len(s)` — O(1)
   Внутри множества хранится отдельное поле (счётчик), которое содержит текущее количество элементов. При добавлении или удалении оно изменяется.

Память: O(n)
Важно: В худшем случае (при большом числе коллизий) операции могут деградировать до O(n)

Применимость и ограничения

Операции с множествами:

```python
a = {1,2,3}
b = {2,3,4}

a & b   # {2,3}     Пересечение — O(min(len(a), len(b)))
a | b   # {1,2,3,4} Объединение — O(len(a) + len(b))
a - b   # {1}       Разность — O(len(a))
a ^ b   # {1,4}     Симметрическая разность — O(len(a) + len(b))
```

Не использовать, если:

- Нужен порядок элементов
- Важны индексы
- Нужен доступ по позиции

Ограничения:

- Множество не сохраняет порядок
- Можно хранить только объекты со стабильным хешем. Потому что если объект изменится после добавления, его хеш изменится, и структура сломается.
- Требует дополнительную память

Компромиссы: скорость поиска достигается за счёт дополнительной памяти (хеш-таблица занимает больше места, чем список)