

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, какие компромиссы он несёт и почему он ведёт себя именно так.

Шаблон раздела структуры данных

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

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. Для каждой операции указывается: синтаксис и оценка времени выполнения и памяти для ключевых операций. Важно для понимания эффективности кода и выбора подходящей структуры под задачу
4. **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

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 \rightarrow вычисление хеша \rightarrow определение индекса в таблице \rightarrow обращение к ячейке.

В Python хеш-таблица не является самостоятельной структурой, доступной пользователю напрямую. Она лежит в основе `set`, `dict`, `Counter`, `defaultdict` и других производных структур.

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
1 hash(x)
```

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

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

3. Проверка:

- Если ячейка пустая \rightarrow объект записывается туда
- Если занята другим объектом (коллизия) \rightarrow применяется поиск следующей подходящей позиции
- Если найден тот же объект \rightarrow 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 \leftrightarrow 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: $8 \rightarrow 16 \rightarrow 32 \rightarrow 64 \rightarrow \dots$

Это упрощает вычисление индекса и управление заполненностью

Resize имеет сложность $O(n)$, но происходит редко, поэтому средняя сложность вставки остаётся $O(1)$ (амортизированная)

Сжатие таблицы

При массовом удалении элементов таблица может не уменьшаться автоматически

Если требуется освободить память, можно создать новую структуру на основе текущей:

```
python
```

```
1 d = dict(d)
2 s = set(s)
```

Это приводит к пересборке таблицы под текущее количество элементов

3. Основные операции и методы

Хеш-таблица поддерживает следующие операции:

1. **Вставка элемента** — $O(1)$
2. **Поиск элемента** — $O(1)$
3. **Удаление элемента** — $O(1)$
4. **Построение из n элементов** — $O(n)$
5. **Расширение таблицы при переполнении (внутренний механизм)** — $O(n)$

В Python хеш-таблица не является самостоятельной структурой, доступной пользователю напрямую, но эти операции реализованы внутри `set` и `dict`

Память: $O(n)$, с дополнительным запасом пустых ячеек + служебные данные

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

4. 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. Хеш-таблица не обеспечивает сортировку по ключам
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)

1. Идея

Множество — это структура данных, которая:

1. Хранит только уникальные элементы
2. Обеспечивает быструю проверку наличия элемента
3. Порядок элементов не гарантируется
4. Поддерживает операции: пересечение, объединение, разность, симметрическая разность

2. Интуитивная модель

Множество основано на хеш-таблице и является её частным случаем (см. раздел «2.1 Хеш-таблица»)

По внутреннему устройству `set` и `dict` очень похожи. Если словарь хранит пары ключ → значение, то множество хранит только ключи без значений.

Принцип работы основан на хешировании элементов

3. Основные операции и методы

Множество поддерживает следующие операции:

1. Создание множества

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

2. Построение из списка длины n

```
python
1 set(nums)      # O(n)
```

3. Проверка принадлежности

```
python
1 x in s         # O(1)
```

4. Добавление элемента

```
python
1 s.add(x)      # O(1)
```

5. Удаление элемента

```
python
1 s.remove(x)   # O(1) ошибка, если x нет
2 s.discard(x)  # O(1) ничего не делает, если x нет
```

6. Получение размера

Внутри множества хранится отдельное поле (счётчик), которое содержит текущее количество элементов. При добавлении или удалении оно изменяется.

```
python
1 len(s)        # O(1)
```

7. Операции над множествами

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

Память: $O(n)$

Важно: В худшем случае операции могут деградировать до $O(n)$ (подробнее см. раздел «2.1 Хеш-таблица»)

4. Применимость и ограничения

Когда использовать:

- Быстрая проверка наличия элемента
- Удаление дубликатов
- Работа с пересечениями / объединениями

Не использовать, если:

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

Ограничения:

- Порядок элементов не гарантируется
- Можно хранить только объекты со стабильным хешем. Потому что если объект изменится после добавления, его хеш изменится, и структура сломается
- Требуется дополнительная память

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