Python: Документация

Барсуков Дмитрий

Февраль 2026

Содержание

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 → вычисление хеша → определение индекса в таблице → 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
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

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
d = dict(d)
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
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 мн-во (не {}, это словарь)
hello_set = set('hello')                      # {'h', 'o', 'l', 'e'}
a = {i ** 2 for i in range(10)}               # {0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
from_list = set([1, 2, 2, 3])                 # {1, 2, 3} — дубликаты убраны
words_set = set(['hi', 'dad', 'hi', 'mum'])  # {'hi', 'dad', 'mum'}
```

2. **Построение из списка длины n**

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

3. **Проверка принадлежности**

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

4. **Добавление элемента**

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

5. **Удаление элемента**

```python
s.remove(x)   # O(1) ошибка, если x нет
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
len(s)        # O(1)
```

7. **Операции над 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))
```

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.
- Требует дополнительную память

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

Словарь (dict)

1. Идея

Словарь — это структура данных, которая:

1. Хранит пары ключ → значение
2. Обеспечивает быстрый доступ к значению по ключу
3. Сохраняет порядок вставки (начиная с Python 3.7)

2. Интуитивная модель

Словарь основан на хеш-таблице (см. раздел «2.1 Хеш-таблица»), но устроен чуть сложнее, чем `set`. Он дополнительно хранит структуру для поддержания порядка вставки, благодаря чему словарь сохраняет порядок вставки.

Словарь можно представить как массив ячеек, где каждая ячейка хранит:

- Ссылку на ключ
- Ссылку на значение

Индекс ячейки вычисляется только по ключу: ключ → hash(ключ) → индекс → проверка ключей через ==, чтобы корректно обработать коллизии. Значение может быть любым объектом.

Добавление, удаление и поиск работают аналогично множеству

3. Основные операции и методы

Словарь поддерживает следующие операции:

1. **Создание словаря**

```python
d = {}                                      # пустой словарь
d = {'dict': 1, 'dictionary': 2}            # {'dict': 1, 'dictionary': 2}
d = dict([(1, 1), (2, 4)])                  # {1: 1, 2: 4}
d = dict(short='dict', long='dictionary')   # {'short': 'dict', 'long': 'dictionary'}
d = {a: a ** 2 for a in range(6)}           # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

2. **Построение словаря из n элементов**

```python
dict(pairs)                 # O(n)
```

3. **Проверка наличия ключа**

```python
key in d                    # O(1)
```

4. **Доступ к значению**
   Если ключа нет — возникает `KeyError`

```python
d[key]                      # O(1)
```

5. **Безопасный доступ**
   Не создаёт новый ключ. Возвращает значение по умолчанию, если ключ отсутствует

```python
d.get(key)                  # O(1)
d.get(key, default_value)   # O(1)
```

6. **Добавление элемента**
   Если ключ уже существует, то значение перезаписывается

```python
d[key] = value              # O(1)
```

7. **Удаление элемента**

```python
del d[key]                  # O(1)
d.pop(key)                  # O(1) удаляет элемент и возвращает его значение
```

8. **Получение размера**
   Внутри словаря хранится отдельное поле (счётчик), которое содержит текущее количество элементов. При добавлении или удалении оно изменяется.

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

9. **View-объекты**
   Это специальные *view-объекты*, не списки. Они не делают копию данных. Их создание — O(1). Они динамические (изменяются при изменении словаря)

```python
# Сложность проверки
key in d.keys()             # O(1)
value in d.values()         # O(n) (нужно перебрать значения)
pair in d.items()           # O(1) (по ключу)
```

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:

- Подсчёт количества
- Группировка данных
- Запоминание индексов
- Быстрый доступ по ключу

Сортировка через подсчёт `(counting sort)` эффективна, когда диапазон значений небольшой

Ограничения:

- Ключ должен иметь стабильный хеш
- Требует дополнительную память

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