

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
2.3	Словарь (dict)	9
3	Паттерны и приёмы решения задач	12
3.1	Скольльзящее окно (sliding window)	12
3.2	Корзины (bucketization)	14

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)

Шаблон раздела “Паттерны и приёмы решения задач”

Каждый паттерн описывается по единому плану, чтобы увидеть общие принципы, работающие в разных задачах, научиться распознавать тип задачи, быстро подбирать подходящий метод и адаптировать их под разные условия, накопить личную копилку проверенных решений.

1. **Идея:** Краткая суть подхода. Что это за приём? Какую проблему решает?
2. **Алгоритм шагами:** Чёткая последовательность действий, которую нужно выполнить, чтобы реализовать этот паттерн
3. **Когда применять (признаки задачи):** Ключевые слова, характерные черты условия, по которым можно опознать, что этот паттерн подходит
4. **Сложность (время / память):** Оценка сложности по времени и памяти в терминах O-большое
5. **Плюсы и минусы подхода:** В каких случаях особенно эффективен, какие ограничения снимает, почему удобен на практике. Когда может быть не лучшим выбором, подводные камни реализации, особенности, о которых нужно помнить
6. **Пример одной задачи:** Конкретная задача с кодом на Python, демонстрирующая применение паттерна

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** — (1)
3. **Удаление элемента** — (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.
- Требуется дополнительная 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
1 d = {}                                # пустой словарь
2 d = {'dict': 1, 'dictionary': 2}      # {'dict': 1, 'dictionary': 2}
3 d = dict([(1, 1), (2, 4)])            # {1: 1, 2: 4}
4 d = dict(short='dict', long='dictionary') # {'short': 'dict', 'long': 'dictionary'}
5 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
1 dict(pairs)                          # O(n)
```

3. Проверка наличия ключа

```
python
1 key in d # 0(1)
```

4. Доступ к значению

Если ключа нет — возникает `KeyError`

```
python
1 d[key] # 0(1)
```

5. Безопасный доступ

Не создаёт новый ключ. Возвращает значение по умолчанию, если ключ отсутствует

```
python
1 d.get(key) # 0(1)
2 d.get(key, default_value) # 0(1)
```

6. Добавление элемента

Если ключ уже существует, то значение перезаписывается

```
python
1 d[key] = value # 0(1)
```

7. Удаление элемента

```
python
1 del d[key] # 0(1)
2 d.pop(key) # 0(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
1 len(d) # 0(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
1 # Сложность проверки
2 key in d.keys()           #  $O(1)$ 
3 value in d.values()       #  $O(n)$  (нужно перебрать значения)
4 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
- Требуется дополнительная память

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

Паттерны и приёмы решения задач

Скольльзящее окно (sliding window)

1. Идея

Скольльзящее окно — приём, когда мы работаем не со всем массивом сразу, а только с ограниченным “кусочком” (подотрезком) вокруг текущей позиции

2. Алгоритм шагами

1. Создать структуру/состояние окна (например `set` , `dict` , счётчики, сумма, максимум и т.д.)
2. Идти по массиву слева направо:
 - Добавить текущий элемент `nums[i]` в состояние окна
 - Если окно стало “слишком большим”, то удалить элемент `nums[i-k]` , который выходит за окно
 - Проверить нужное условие внутри окна (что именно проверяем — зависит от задачи)
3. Если ни на одном шаге условие не выполнилось — вернуть `False` / нужный ответ

Ключевой смысл: мы не пересчитываем всё заново, а обновляем состояние “на лету”

3. Когда применять (признаки задачи)

Чаще всего условие в задаче звучит так:

- Учитывать только элементы на расстоянии не больше `k` по индексам
- Сравнивать только последние `k` элементов
- Элементы должны быть в пределах окна длины `k`
- Нужно проверять/поддерживать свойство на “локальном участке”, который двигается

4. Сложность (время / память)

Время: $O(n)$ — каждый элемент один раз добавляется и один раз удаляется из структуры

Память: $O(k)$ — храним состояние только для окна размера до `k`

Важно: это верно, если операции обновления состояния в среднем $O(1)$ (например, `set` , `dict`) или $O(\log k)$ (например, балансирующее дерево/heap — тогда будет $O(n \log k)$).

5. Плюсы и минусы подхода

Плюсы:

- Часто превращает $O(n^2)$ (двойные циклы) в $O(n)$
- Память ограничена размером окна $O(k)$, а не всем n
- Подходит для потоковой обработки данных — алгоритм обрабатывает элементы последовательно и хранит только состояние окна, не требуя всего массива целиком

Минусы:

- Нужно правильно поддерживать состояние окна (легко ошибиться в “что удалять”)
- Если структура состояния дорогая (например, нужно поддерживать сортировку), то время будет не $O(n)$, а $O(n \log k)$

6. Пример одной задачи

LeetCode 219: Contains Duplicate II

Найти два одинаковых значения `nums[i] == nums[j]`, такие что индексы близко: расстояние между индексами не больше `k`. То есть нас интересуют только пары внутри “окна” длины `k`.

Идея через окно: На позиции `i` нам достаточно знать, встречалось ли `nums[i]` среди последних `k` элементов. Значит, держим `set` текущего окна

```
python
1 s = set() # элементы текущего окна (последние k значений)
2
3 for i, v in enumerate(nums):
4     if v in s:
5         return True
6
7     s.add(v)
8
9     # держим окно размера <= k
10    if i >= k:
11        s.remove(nums[i - k])
12
13 return False
```

Время: $O(n)$

Память: $O(k)$

Корзины (bucketization)

1. Идея

Корзины — это приём, при котором мы разбиваем пространство значений на интервалы (группы) фиксированной ширины и кладём каждый элемент в соответствующую «ячейку».

Главная цель — сильно сузить круг кандидатов для проверки, чтобы не сравнивать новый элемент со всеми предыдущими.

Если два числа должны быть «близки» по значению (например, $|a - b| \leq t$), то:

- либо они попадут в одну корзину
- либо в соседние корзины

Таким образом, при добавлении нового числа мы проверяем только 1–3 корзины, а не весь массив

2. Алгоритм шагами

1. Выбрать ширину корзины w

2. Для каждого числа v :

- Вычислить номер корзины:

```
python
1 bucket_id = v // w
```

- Проверить: текущую корзину, левую / правую соседнюю
- Если найден подходящий элемент, то вернуть ответ, иначе добавить v в свою корзину

3. Когда применять (признаки задачи)

Корзины подходят, когда есть условие близости по значению:

- $|a - b| \leq t$
- расстояние $\leq r$

4. Сложность (время / память)

Если в каждой корзине хранится не более одного элемента:

1. **Время:** $O(n)$
2. **Память:** $O(n)$

Если корзины могут содержать много элементов, то время может вырасти до $O(n^2)$

5. Плюсы и минусы подхода

Плюсы:

- Часто превращает $O(n^2)$ (двойные циклы) в $O(n)$
- Очень эффективно для числовых данных
- Простая арифметическая реализация

Минусы:

- Работает только там, где есть геометрический / числовой смысл близости
- Требуется аккуратного выбора ширины корзины

6. Пример одной задачи

LeetCode 220: Contains Duplicate III

Найти два числа, такие что:

1. $|\text{nums}[i] - \text{nums}[j]| \leq \text{valueDiff}$
2. $|i - j| \leq \text{indexDiff}$

Идея через корзины:

- Разбиваем числовую ось на корзины $\text{valueDiff} + 1$. Проверяем текущую и соседние корзины
- Дополнительно ограничиваем размер хранимых элементов через скользящее окно (по индексам)

python

```
1 d = {}
2 w = valueDiff + 1
3 for i, v in enumerate(nums):
4     new_id = v // w
5
6     if new_id in d: return True
7
8     if new_id-1 in d and v - d[new_id - 1] <= valueDiff: return True
9     if new_id+1 in d and d[new_id + 1] - v <= valueDiff: return True
10
11     d[new_id] = v
12
13     if i >= indexDiff:
14         del_id = nums[i - indexDiff] // w
15         del d[del_id]
16 return False
```

Время: $O(n)$

Память: $O(k)$