Методичка: структуры данных Python

list • tuple • dict • set

Дата: 17.02.2026 | Формат: 2 пары по 80 минут

Эта методичка помогает быстро и уверенно выбирать структуру данных под задачу. Фокус: синтаксис, мутабельность, порядок, уникальность, ключевые методы и оценка Big O (средний и худший случай).

Как выбирать структуру данных (быстрая шпаргалка)

Нужен порядок + индексы → list Нужно “записать” фиксированные значения и запретить изменения → tuple Нужен быстрый доступ по ключу/ID → dict Нужна уникальность и быстрый “x in ...” → set

Сравнение в одной таблице

Тип	Скобки / создание	Мутабельность	Порядок	Повторы	Доступ
list	[] или list()	да	да	да	по индексу
tuple	() или tuple()	нет	да	да	по индексу
dict	{} или dict()	да	порядок вставки*	ключи уникальны	по ключу
set	set() или {1,2}	да	нет (логически)	нет	membership / операции

* В Python 3.7+ dict сохраняет порядок вставки, но использовать его нужно ради ключей/доступа, а не ради сортировки.

1) list — список

Коротко: упорядоченная коллекция, мутабельная, поддерживает индексы и повторы.

```
nums = [10, 20, 20]
nums.append(30)
nums[0] = 999
print(nums)  # [999, 20, 20, 30]
```

Ключевые свойства

Скобки / создание	[] или list(iterable)
Мутабельность	Да (можно менять элементы, добавлять, удалять)
Порядок	Да (элементы идут в том порядке, как добавлялись)
Повторы	Да
Индексы	Да (0..n-1)
Типы внутри	Любые типы (в т.ч. смешанные), но лучше держать “один смысл”

Методы (основные)

Метод	Что делает	Мини-пример
append(x)	Добавить x в конец.	a.append(5)
extend(iter)	Добавить все элементы из iter в конец.	a.extend([4,5])
insert(i, x)	Вставить x по индексу i (сдвигает хвост).	a.insert(0, 99)
pop([i])	Удалить и вернуть элемент (по умолчанию последний).	x = a.pop()
remove(x)	Удалить первое вхождение x.	a.remove(2)
clear()	Очистить список.	a.clear()
index(x)	Индекс первого x (ошибка если нет).	i = a.index(7)
count(x)	Сколько раз x встречается.	c = a.count(1)
sort()	Отсортировать на месте.	a.sort()
reverse()	Развернуть на месте.	a.reverse()

Big O для list (CPython, оценка):

Операция	Average	Worst	Комментарий
Доступ по индексу a[i]	O(1)	O(1)	Прямая адресация по индексу.
Присваивание a[i]=x	O(1)	O(1)	Замена элемента по индексу.
append(x)	O(1)*	O(n)	Амортизированно O(1); иногда происходит перераспределение массива.
pop() (с конца)	O(1)	O(1)	Удаление последнего без сдвигов.
insert(i,x)	O(n)	O(n)	Нужно сдвинуть элементы вправо.
pop(i) / del a[i]	O(n)	O(n)	Нужно сдвинуть хвост влево.
x in a (поиск)	O(n)	O(n)	Линейный проход, сравнение по очереди.

* Амортизированно: в среднем быстро, но иногда дороже из-за расширения внутреннего массива.

Плюсы и минусы

Плюсы: порядок, индексы, удобно хранить “список карточек”, легко добавлять в конец. Минусы: поиск “x in list” медленнее на больших данных; вставки/удаления в начале/середине дорогие.

Когда выбирать

Вам важен порядок (например: список товаров/сообщений/оценок). Нужно обращаться по индексу (0-й, 1-й… элемент). Нужно хранить повторы (оценки, результаты, логи).

2) tuple — кортеж

Коротко: как list, но неизменяемый. Порядок и индексы есть, повторы допустимы.

```
user = ("Ali", 16, "Almaty")
name, age, city = user
print(name, age, city)
```

Ключевые свойства

Скобки / создание	() или tuple(iterable)
Мутабельность	Нет (после создания менять нельзя)
Порядок	Да
Повторы	Да
Индексы	Да
Типы внутри	Любые (часто используют как “запись” из нескольких полей)

Методы tuple

Метод	Что делает	Мини-пример
count(x)	Сколько раз x встречается.	t.count(5)
index(x)	Индекс первого x.	t.index("Ali")

Big O для tuple:

Операция	Average	Worst	Комментарий
Доступ по индексу t[i]	O(1)	O(1)	Как у list, потому что это тоже последовательность.
x in t (поиск)	O(n)	O(n)	Линейный проход.
Создание tuple из iterable	O(n)	O(n)	Нужно пройти все элементы.

Плюсы и минусы

Плюсы: неизменяемость (безопасность), можно использовать как ключ в dict (если элементы хэшируемые). Минусы: нельзя дополнять/удалять элементы; для “живых” коллекций неудобно.

Когда выбирать

Нужны фиксированные данные: координата (x, y), запись (name, age, city). Хотите защитить данные от изменений.

3) dict — словарь (ключ → значение)

Коротко: хранит пары ключ → значение. Доступ по ключу обычно очень быстрый.

```
student = {"name": "Dana", "age": 15}
student["age"] = 16
print(student.get("city", "—"))  # —
```

Ключевые свойства

Скобки / создание	{} или dict()
Мутабельность	Да
Порядок	Сохраняет порядок вставки (Python 3.7+), но смысл dict — ключи и доступ.
Ключи	Уникальные, должны быть хэшируемыми (int, str, tuple и т.д.).
Значения	Любые типы.
Типичный кейс	id → объект, имя → возраст, слово → частота.

Методы (самые нужные)

Метод	Что делает	Мини-пример
get(k, default)	Вернуть значение по ключу, иначе default (без ошибки).	age = d.get("age", 0)
keys()	Представление ключей.	for k in d.keys(): ...
values()	Представление значений.	for v in d.values(): ...
items()	Пары (k, v) — лучший способ перебора.	for k,v in d.items(): ...
pop(k)	Удалить ключ и вернуть значение.	v = d.pop("age")
update(other)	Добавить/обновить из другого dict.	d.update({"x":1})
setdefault(k, default)	Если ключа нет — создать его.	d.setdefault("cat", []).append(x)
clear()	Очистить словарь.	d.clear()

Big O для dict (хеш-таблица):

Операция	Average	Worst	Комментарий
Получить d[k]	O(1)	O(n)	Среднее O(1). Худший случай редок (коллизии/перестроение).
Записать d[k]=v	O(1)	O(n)	Среднее O(1); иногда расширение таблицы.
Удалить del d[k]	O(1)	O(n)	Среднее O(1).
k in d (проверка ключа)	O(1)	O(n)	Очень быстро по хэшу.
Перебор items/keys/values	O(n)	O(n)	Нужно пройти все элементы.

Плюсы и минусы

Плюсы: лучший выбор для “найти по ключу/ID”; удобно хранить свойства объекта; частотные таблицы. Минусы: память дороже, чем у list; ключи должны быть хэшируемыми; не подходит, если нужен доступ по индексу.

Когда выбирать

Нужно быстро: “дай пользователя по id”, “дай цену по названию”. Нужно хранить атрибуты объекта (name, age, city). Нужно считать частоты (слово → количество).

4) set — множество

Коротко: уникальные элементы без логического порядка. Очень быстро отвечает на вопрос: “x есть?”

```
nums = [1,1,2,3,3]
uniq = set(nums)
print(2 in uniq)  # True
```

Ключевые свойства

Создание	set() для пустого; {1,2,3} для непустого
Мутабельность	Да (set). Есть также frozenset — неизменяемый вариант.
Порядок	Логически нет (не используйте set ради порядка).
Повторы	Нет (автоматически убираются).
Элементы	Должны быть хэшируемыми (нельзя list/dict/set внутрь).
Типичный кейс	уникальность, фильтры, пересечения, разности, быстрый membership.

Методы и операции

Метод	Что делает	Мини-пример
add(x)	Добавить элемент.	s.add(5)
remove(x)	Удалить элемент (ошибка если нет).	s.remove(5)
discard(x)	Удалить без ошибки, если нет.	s.discard(5)
pop()	Удалить и вернуть “какой-то” элемент.	x = s.pop()
union / |	Объединение.	a | b
intersection / &	Пересечение.	a & b
difference / -	Разность.	a - b
symmetric_difference / ^	Симметрическая разность.	a ^ b
issubset / <=	Подмножество.	a <= b
issuperset / >=	Надмножество.	a >= b
clear()	Очистить.	s.clear()

Big O для set (хеш-таблица):

Операция	Average	Worst	Комментарий
x in s (membership)	O(1)	O(n)	Обычно очень быстро.
add(x)	O(1)	O(n)	Среднее O(1); иногда расширение.
remove/discard	O(1)	O(n)	Среднее O(1).
Union a|b	O(len(a)+len(b))	O(len(a)+len(b))	Нужно пройти элементы и сложить.
Intersection a&b;	O(min(len(a),len(b)))	O(len(a)*len(b))	На практике близко к min(...) из-за хешей.
Difference a-b	O(len(a))	O(len(a)+len(b))	Проходим a и проверяем отсутствие в b.

Плюсы и минусы

Плюсы: уникальность “из коробки”; удобно для пересечений/разностей; быстрый “x in ...”. Минусы: нет логического порядка; нельзя хранить изменяемые типы внутрь (list/dict/set).

Когда выбирать

Нужно убрать дубликаты. Нужно быстро проверять принадлежность. Нужно сравнить два набора: общие элементы, отличия, объединение.

Дополнение: frozenset

Это “замороженное” множество: как set, но неизменяемое. Полезно, когда набор должен быть ключом в dict или гарантированно не меняться.

```
fs = frozenset([1,1,2])
# fs.add(3)  # нельзя
```

Итоги и выбор структуры под задачу

Ниже — практичная таблица “сценарий → структура”.

Сценарий	Лучший выбор	Почему
Нужен порядок, работа по индексам	list / tuple	Индексы и последовательность.
Нужен быстрый поиск по ключу (id → объект)	dict	Доступ по ключу обычно O(1).
Нужно быстро проверять “x есть?”	set (или ключи dict)	Membership обычно O(1).
Нужно хранить уникальные элементы	set	Дубликаты автоматически убираются.
Нужно “записать” фиксированные значения и не менять	tuple / frozenset	Неизменяемость = безопасность.

Примечание по Big O

Для dict и set оценки O(1) — это средний случай. Худший случай O(n) возможен теоретически, но в учебных задачах и большинстве проектов ориентируются на average-case.