Методичка для студентов

Урок 2: Структуры данных + повтор ООП (Python для AI)

Цель урока: научиться выбирать структуры данных под задачу подготовки данных и закрепить ООП как способ упаковки логики в понятные сущности (Dataset, Preprocessor, Analyzer).

Формат: 2 пары по 80 минут. Уровень: начальный/смешанный.

Как это связано с AI

- В AI-проектах большая часть работы - это данные: сбор, чистка, группировка, подсчет статистики, удаление дублей.
- Структуры данных задают скорость и удобство обработки: где-то нужен порядок (list/tuple), где-то быстрый поиск (set/dict).
- ООП помогает держать код масштабируемым: когда появляются сущности и состояние (настройки, параметры, результаты).

Часть 1. Структуры данных Python

В Python основные структуры данных, которые мы используем постоянно:

- list - список: порядок важен, можно менять.
- tuple - кортеж: порядок важен, но менять нельзя.
- dict - словарь: ключ - значение, быстрый доступ по ключу.
- set - множество: только уникальные элементы, быстрый поиск и операции множеств.

1) list - список

Список - упорядоченная коллекция элементов. Список изменяемый: можно добавлять, удалять, менять элементы.

```
nums = [10, 20, 30]
nums.append(40)      # добавить в конец
nums[0] = 99         # изменить по индексу
print(nums)          # [99, 20, 30, 40]
```

Где list подходит лучше всего:

- Нужно сохранить порядок данных (временной ряд, список сообщений, последовательность шагов).
- Нужно накапливать элементы (append).
- Нужен доступ по индексу.

Типичные операции и скорость (идея):

Операция	Пример	Скорость (смысл)
Доступ по индексу	nums[i]	быстро (O(1))
Добавить в конец	nums.append(x)	обычно быстро (амортиз. O(1))

Поиск элемента	x in nums	медленнее на больших данных (O(n))
Вставка/удаление в середине	insert/pop(i)	медленнее (сдвиг элементов)

Важно: если вы часто делаете поиск в большом списке, возможно, вам нужен set или dict.

2) tuple - кортеж

Кортеж похож на список, но он неизменяемый. Это удобно, когда данные должны оставаться константными.

```
point = (10, 5)
print(point[0])  # 10
# point[0] = 99  # ошибка: tuple нельзя менять
```

Когда tuple лучше, чем list:

- Данные не должны меняться (координаты, настройки, параметры).
- Нужно использовать значение как ключ в dict или элемент set (tuple обычно хешируемый).

Пара (x, y), RGB-цвет (r, g, b), размер (width, height) - типичные примеры.

3) dict - словарь

Словарь хранит пары ключ -> значение. Ключи должны быть уникальными.

```
student = {"name": "Ali", "age": 16}
print(student["name"])  # Ali
student["age"] += 1
```

Словари - базовый инструмент для AI-подготовки данных:

- подсчет частот (сколько раз встречается элемент);
- группировка (category -> список элементов);
- быстрые проверки наличия ключа (O(1) в среднем).

Частота чисел/слов через dict.get:

```
nums = [5, 5, 2, 2, 2, 3, 3, 5, 1, 1, 1]
freq = {}

for x in nums:
    freq[x] = freq.get(x, 0) + 1

print("freq:", freq)
```

Идея скорости:

- x in freq (проверка ключа) обычно быстро.
- freq[x] чтение/обновление обычно быстро.

4) set - множество

Множество хранит только уникальные элементы. Порядок не гарантируется.

```
nums = [1, 2, 2, 3, 3, 3]
unique = set(nums)
print(unique)  # {1, 2, 3}
```

Главные случаи, когда set нужен прямо сейчас:

- Убрать дубли и быстро узнать количество уникальных элементов.
- Быстрый поиск: x in set обычно быстрее, чем x in list на больших данных.
- Операции множеств: пересечение, объединение, разность.

```
A = {"python", "sql", "ml"}
B = {"python", "react"}

print(A & B)  # пересечение
print(A | B)  # объединение
print(A - B)  # разность
```

Ограничение: в set нельзя класть изменяемые типы (например, list или dict). Можно класть int/str/tuple.

Вложенные структуры данных

В реальных задачах данные почти всегда вложенные: список словарей, словарь списков, словарь словарей и т.д.

1) Список словарей (records)

Часто данные приходят как «таблица», где каждая строка - это словарь.

```
users = [
    {"id": 1, "name": "Ali", "age": 16, "city": "Almaty"},
    {"id": 2, "name": "Dana", "age": 15, "city": "Astana"},
]
```

Плюсы: легко читать, удобно фильтровать и преобразовывать.

2) Словарь списков (columns)

Иногда удобно хранить данные по колонкам (похоже на будущий DataFrame):

```
data = {
    "age": [16, 15, 16, 14],
    "city": ["Almaty", "Almaty", "Astana", "Astana"],
}
```

3) Словарь словарей (index)

Когда нужен быстрый доступ по имени/идентификатору:

```
grades = {
    "Ali": {"math": 90, "eng": 80, "cs": 95},
    "Dana": {"math": 100, "eng": 70, "cs": 85},
}
```

4) Список словарей с вложенными списками

Типичный формат заказа/документа: внутри есть список элементов.

```
orders = [
  {"client": "Ali", "items": [{"title":"Burger","price":2000}, {"title":"Cola","price":500}]},
  {"client": "Dana", "items": [{"title":"Pizza","price":3500}]},
]
```

Практический паттерн: проход по двум уровням вложенности

```
total = 0
for order in orders:
    for item in order["items"]:
        total += item["price"]
print("total:", total)
```

Как выбирать структуру данных

Выбор структуры - это про 3 вещи: порядок, уникальность, скорость доступа.

Шпаргалка выбора

Если нужно...	Чаще всего берём...
Сохранять порядок, накапливать элементы	list

Сохранить порядок и запретить изменения	tuple
Быстро искать/обновлять по ключу	dict
Уникальность и быстрый поиск + операции множеств	set

Типичные ошибки выбора

- Хранить уникальные значения в list и постоянно проверять x in list (будет медленно на больших данных).
- Смешивать структуру данных «как попало»: вместо понятных records делать списки несвязанных значений.
- Создавать dict там, где ключи не нужны (лишняя сложность).

Практический ориентир: сначала сделайте понятное решение, потом ускоряйте (обычно через set/dict).

Часть 2. Повтор ООП

ООП - это способ описывать программу через сущности (объекты), у которых есть состояние (данные) и поведение (методы).

Класс, объект, атрибуты, методы

```
class Student:
    def __init__(self, name, grades):
        self.name = name
        self.grades = grades

    def avg(self):
        return sum(self.grades) / len(self.grades)

ali = Student("Ali", [90, 80, 70])
print(ali.name)
print(ali.avg())
```

Где ООП используется в реальных проектах

- Когда есть несколько сущностей: Dataset, Preprocessor, Model, Trainer, Evaluator.
- Когда у сущностей есть параметры и состояние: настройки, метрики, кеши, результаты.
- Когда проект растёт и важно разделять ответственность.

Когда ООП лучше НЕ использовать

- Очень маленький скрипт на 20-40 строк, который запускается один раз.
- Класс создаётся «для красоты», но в нём 1 метод без состояния - чаще проще функция.
- В ноутбуке на раннем этапе исследования данных - сначала прототип, потом архитектура.

Здоровое правило: простой код сначала, ООП - когда появилась сложность и повторяемость.

4 столпа ООП

1) Инкапсуляция

Скрываем внутренние детали и даём безопасные методы для работы.

```
class Bank:
    def __init__(self):
        self._balance = 0  # внутреннее поле

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount

    def get_balance(self):
        return self._balance
```

2) Наследование

Создаём общий базовый класс и специализированные версии.

```
class Model:
    def predict(self, x):
        raise NotImplementedError

class SumModel(Model):
    def predict(self, x):
        return sum(x)
```

3) Полиморфизм

Один и тот же интерфейс, разная реализация.

```
def run(model, x):
    return model.predict(x)

print(run(SumModel(), [1, 2, 3]))  # 6
```

4) Абстракция

Оставляем главное, детали прячем за интерфейсом (например, transform/predict).

```
class Preprocessor:
    def transform(self, text):
        raise NotImplementedError

class LowerPreprocessor(Preprocessor):
    def transform(self, text):
        return text.lower()
```

Мини-pipeline обработки текста (пример из урока)

Ниже - цельный пример, который связывает структуры данных и ООП. Он без библиотек: только базовый Python.

Компоненты

- TextDataset хранит список строк (list).
- Preprocessor чистит текст: lower + нормализация пробелов.
- Analyzer считает частоты (dict) и уникальные слова (set).

Полный пример кода

```
class TextDataset:
    def __init__(self, texts):
        self.texts = texts

    def add(self, text):
        self.texts.append(text)

    def get_all(self):
        return self.texts


class Preprocessor:
    def transform(self, text):
        raise NotImplementedError


class BasicPreprocessor(Preprocessor):
    def transform(self, text):
        text = text.lower()
        text = " ".join(text.split())   # убираем лишние пробелы
        return text


class Analyzer:
    def __init__(self):
        self.freq = {}
        self.unique = set()

    def fit(self, texts):
        self.freq = {}
        self.unique = set()
```

```
        for text in texts:
            for w in text.split():
                self.unique.add(w)
                self.freq[w] = self.freq.get(w, 0) + 1

    def top_word(self):
        best_word = None
        best_count = -1
        for word, count in self.freq.items():
            if (
                count > best_count
                or (count == best_count and (best_word is None or word < best_word))
            ):
                best_word = word
                best_count = count
        return best_word, best_count


# --- Demo ---
dataset = TextDataset([
    "AI is cool",
    "AI   is   useful useful",
    "Python is cool"
])

prep = BasicPreprocessor()
clean_texts = [prep.transform(t) for t in dataset.get_all()]

an = Analyzer()
an.fit(clean_texts)

print("unique:", len(an.unique))
print("freq:", an.freq)
print("top:", an.top_word())
```

Что здесь важно заметить

- Структуры данных: list (texts), dict (freq), set (unique).
- ООП даёт понятные роли: где хранятся данные, где чистка, где анализ.
- Если понадобится расширение - можно добавить новый Preprocessor (например, удаление пунктуации) без переписывания Analyzer.

Чек-лист для самопроверки

- Могу объяснить разницу list/tuple/dict/set и привести пример применения.
- Понимаю, почему x in list медленнее, чем x in set на больших данных.
- Умею читать вложенные структуры (list[dict], dict[list], dict[dict]).
- Знаю 4 столпа ООП и могу показать мини-пример каждого.
- Могу собрать простой pipeline: Dataset -> Preprocess -> Analyze.