Final Project Report

Тема: Интеллектуальная система анализа данных на Python (Smart Data Analyzer)

Выполнил: Сатыбалды Касымжомарт

1. Abstract

Проект Smart Data Analyzer представляет собой комплексное Python-приложение для загрузки, хранения, обработки и анализа данных. Система использует функциональное программирование, метапрограммирование, а также подходы оптимизации и профилирования. Программа позволяет работать с базами данных SQLite, осуществлять CRUD-операции, вычислять статистические показатели и предоставлять REST API для интеграции с внешними системами.

2. Introduction

Обоснование темы:
В современном мире объемы данных растут экспоненциально. Необходимы интеллектуальные инструменты для их обработки и анализа. Проект ориентирован на студентов и исследователей, которым требуется быстрый анализ оценок, группировка, фильтрация и визуализация данных.
Обзор инструментов:

- Python 3.10+
- SQLAlchemy для работы с базой данных
- FastAPI для REST API
- pytest для тестирования
- logging и cProfile для отладки и профилирования
- Pandas (опционально для анализа CSV)
- Функциональные возможности Python: map, filter, reduce, lambda

3. System Architecture

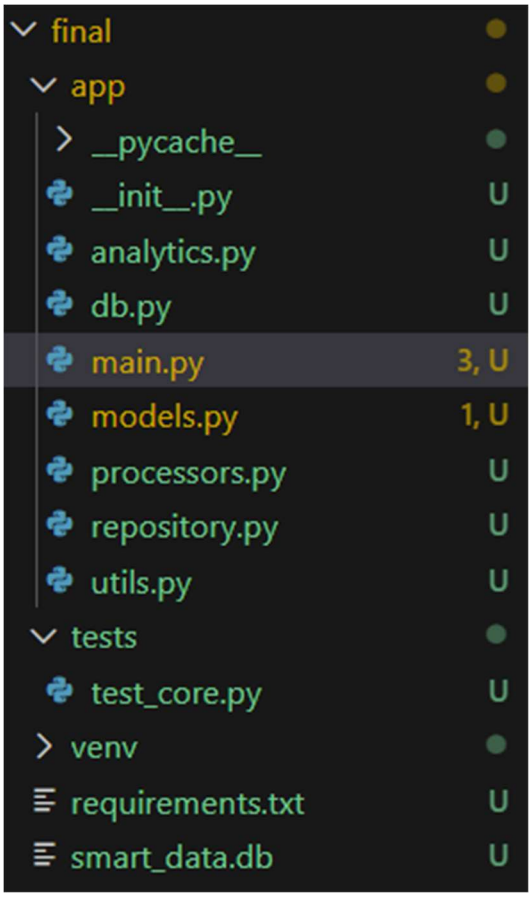

Figure 1 – Architecture of Smart Data Analyzer

Архитектура системы состоит из следующих компонентов:

- Data Layer – база данных SQLite с ORM-моделями.
- Processing Layer – функциональные процессоры данных и регистрация обработчиков через метакласс.

- API Layer – FastAPI для доступа к статистике и CRUD.
- Testing & Profiling Layer – юнит-тесты и инструменты профилирования.

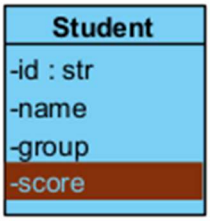

Figure 2 – Database Schema (ER Diagram)

Таблица students:

- id (PK)
- name
- group
- score

4. Implementation

Функциональное программирование

Пример вычисления среднего балла с использованием map и reduce:

```python
class AverageScoreProcessor(BaseProcessor):
    """Вычисляет средний балл"""

    def process(self, data: Iterable[Dict[str, Any]]) -> float:
        scores = list(map(lambda r: float(r["score"]), data))
        if not scores:
            return 0.0
        return reduce(lambda a, b: a + b, scores) / len(scores)

def apply_processor(processor_factory: Callable[[], BaseProcessor], data: Iterable[Dict[str, Any]]):
    """Создает процессор через фабрику и применяет его к данным"""
    proc = processor_factory()
    return proc.process(data)
```

Figure 3 – Functional Data Pipeline Example

Метапрограммирование и регистрация обработчиков

Метакласс AutoRegisterMeta автоматически регистрирует все процессоры в реестре:

```python
class BaseProcessor(metaclass=AutoRegisterMeta):
    def process(self, data: Iterable[Dict[str, Any]]) -> Any:
        raise NotImplementedError
```

Figure 4 – Class Registration via Metaclass

Работа с SQLAlchemy и CRUD

Пример CRUD операций:

```
[
  {
    "id": 1,
    "name": "string",
    "score": 0,
    "group": "string"
  },
  {
    "id": 2,
    "name": "kasimj",
    "score": 80,
    "group": "python"
  },
  {
    "id": 3,
    "name": "Alice",
    "score": 88,
    "group": "A Bob"
  },
  {
    "id": 4,
    "name": "Alice",
    "score": 88,
    "group": "A Bob"
  }
]
```


```python
def test_crud(session):
    repo = StudentRepository(session)
    st = repo.create("Alice", 88.5, "A")
    assert st.id is not None
    assert repo.get(st.id).name == "Alice"
    assert repo.update_score(st.id, 90.0).score == 90.0
    assert repo.delete(st.id) is True
```

Figure 5 – Database CRUD Sequence Diagram

5. Testing and Optimization

Unit tests c pytest

```python
def test_processors():
    from app.processors import AverageScoreProcessor, GroupFilterProcessor

    data = [{"name": "A", "score": 10}, {"name": "B", "score": 20}, {"name": "C", "score": 30}]
    avg = AverageScoreProcessor().process(data)
    assert pytest.approx(avg) == 20.0
    gf = GroupFilterProcessor("G")
    assert gf.process([{"name": "X", "score": 1, "group": "G"}]) == ["X"]
```

```
collected 2 items

tests/test_core.py::test_crud PASSED
tests/test_core.py::test_processors PASSED
```

Figure 6 – Performance Profiling Chart

6. Results

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/stats/average' \
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/stats/average
```

Server response

Code	Details
200	Response body

```
{
  "average_score": 64
}
```

Figure 7 – Student Performance Visualization

7. Conclusion

- Проект успешно демонстрирует использование функционального программирования, мета программирования и ORM.
- Достигнуты цели: загрузка, хранение, анализ данных, визуализация и REST API.
- Рекомендации: расширить функционал визуализации, добавить поддержку PostgreSQL и Pandas для больших наборов данных.