Техническое Задание: Анализатор Лог-Файлов Веб-Сервера

1. Общее Описание

Разработать консольное приложение на Python для анализа лог-файлов веб-сервера в формате **Combined Log Format**. Приложение должно эффективно обрабатывать большие файлы, извлекать статистику и генерировать финальный HTML-отчет с использованием Jinja2.

2. Структура Проекта

```
log-analyzer/
├── access.log        # Исходный файл с логами (заглушка)
├── report_template.html # HTML шаблон для отчета (Jinja2)
├── main.py           # Основной исполняемый файл и логика
└── utils.py          # Вспомогательные классы (Дескриптор, Декоратор)
```

3. Требования к Функциональности (Модули и Классы)

3.1. Модуль utils.py (Вспомогательные Инструменты)

3.1.1. Дескриптор StatusCodeValidator

- **Назначение:** Обеспечить, что атрибут status_code в классе LogEntry является валидным кодом ответа HTTP.
- **Методы:**
  - __set_name__(self, owner, name): Стандартный метод дескриптора для сохранения имени атрибута.
  - __set__(self, obj, value):
    - Пытается преобразовать value в **целое число**.
    - Проверяет, что число находится в диапазоне **100** до **599**.
    - В случае успеха устанавливает значение.

        - В случае ошибки преобразования или невалидного диапазона, устанавливает значение 0 (как признак ошибки парсинга) и выводит предупреждение.

3.1.2. Декоратор @timing

- **Назначение:** Измерять и логировать время выполнения метода, к которому он применен.
- **Функционал:**
    - При вызове оборачиваемой функции, фиксирует время **до** и **после** выполнения.
    - Выводит в консоль имя функции и затраченное время в секундах (например, Метод 'analyze' выполнен за 1.2345 сек.).
    - Сохраняет измеренное время в атрибут analysis_time первого аргумента (экземпляра класса LogAnalyzer).

3.2. Модуль main.py (Основная Логика)

3.2.1. Класс FileContextManager (Менеджер Контекста)

- **Назначение:** Безопасно открывать и закрывать лог-файл, обрабатывая исключения.
- **Методы:**
    - __init__(self, file_path, mode='r', encoding='utf-8'): Принимает путь к файлу.
    - __enter__(self):
        - Открывает файл по указанному пути.
        - Обрабатывает исключение FileNotFoundError.
        - Возвращает **объект файла**.
    - __exit__(self, exc_type, exc_val, exc_tb):

        - Гарантирует **закрытие** файла.
        - Обрабатывает, если возникли исключения внутри блока with (возвращает False, чтобы исключение было поднято).

3.2.2. Класс LogEntry

- **Назначение:** Хранить данные, извлеченные из одной строки лога, и предоставлять удобный доступ к ним.
- **Атрибуты:** ip, timestamp, method, url, protocol, status_code (с дескриптором), size.
- **Методы:**
    - __init__(...): Конструктор для инициализации атрибутов.
    - @classmethod from_log_line(cls, line):
        - Использует **регулярное выражение** (LOG_PATTERN) для парсинга строки лога.
        - В случае успешного совпадения создает и возвращает новый экземпляр LogEntry.
        - В случае неудачи возвращает None.
    - @property request_path(self):
        - Возвращает **чистый путь** из полного URL (например, /index.html из GET /index.html HTTP/1.0).

3.2.3. Генератор log_parser(file_path)

- **Назначение:** Эффективно построчно читать лог-файл и генерировать объекты LogEntry.
- **Функционал:**
    - Использует **FileContextManager** для открытия файла.
    - Итерирует по строкам файла.

- Для каждой строки вызывает LogEntry.from_log_line().
- Если получен объект LogEntry, **генерирует** его (yield).
- Игнорирует или логирует строки, которые не удалось распарсить.

3.2.4. Класс LogAnalyzer

- **Назначение:** Управлять процессом анализа, хранить результаты и генерировать отчет.
- **Атрибуты (Для хранения результатов):**
  - self.ip_counts (Counter): Подсчет IP-адресов.
  - self.url_counts (Counter): Подсчет запрошенных URL.
  - self.status_counts (Counter): Подсчет кодов ответов.
  - self.req_by_hour (defaultdict): Группировка запросов по часу (ключ: час (0-23), значение: Counter или int).
  - self.heavy_requests (deque): deque(maxlen=10) для хранения **10** самых "тяжелых" (по полю size) запросов.
  - self.analysis_time (float): Время выполнения анализа (устанавливается декоратором).
- **Методы:**
  - __init__(self, file_path): Сохраняет путь к файлу. Инициализирует все атрибуты для хранения результатов.
  - @timing analyze(self):
    - Вызывает **генератор** log_parser для получения объектов LogEntry.
    - В цикле проходит по всем сгенерированным записям:
      - **Обновляет** все Counter и defaultdict (статистика).
      - **Проверяет**, является ли запрос "тяжелым" (например, size > 1024). Если да, добавляет его в deque. Заполнять его до лимита.

- generate_report(self, template_path, output_path='report.html'):
  - Инициализирует окружение **Jinja2**.
  - Загружает шаблон report_template.html.
  - Формирует словарь данных для шаблона, включая:
    - top_10_urls: self.url_counts.most_common(10)
    - status_summary: Сводка по 2xx, 4xx, 5xx.
    - heavy_list: Список из self.heavy_requests.
    - time_taken: self.analysis_time.
  - Рендерит шаблон и **сохраняет** результат в output_path.

4. Требования к Внешнему Виду Отчета (Jinja2)

- **Шаблон report_template.html** должен быть прост, но информативен.
- Он должен содержать следующие разделы:
  1. Заголовок и время анализа.
  2. Таблица **"Топ-10 самых посещаемых URL"**.
  3. График или таблица **"Распределение кодов ответа"** (2xx, 3xx, 4xx, 5xx).
  4. Раздел **"Самые тяжелые запросы"** (список из deque).

5. Порядок Выполнения (Финальный Скрипт main.py)

1. Создать экземпляр LogAnalyzer.
2. Вызвать analyzer.analyze().
3. Вызвать analyzer.generate_report().