Мидтерм проект

Тема проекта:

Разработка высокопроизводительного Python-приложения для управления задачами с использованием продвинутых возможностей языка.

Выполнил:
Сатыбалды Касымжомарт

Введение

Цели:

Создаётся система планировщика задач (task manager), которая:

- хранит задачи с разными приоритетами (используя dataclasses и collections),
- позволяет обрабатывать данные итераторами и генераторами,
- использует декораторы для логирования и метаклассы для контроля структуры классов,
- разделяется на модули и пакеты для удобства масштабирования,
- обрабатывает ошибки через исключения,
- поддерживает асинхронное выполнение (например, скачивание данных из сети),
- умеет распределять нагрузку через многопоточность и мультипроцессинг.

Почему именно продвинутые возможности Python

Современные приложения требуют высокой производительности и гибкости. Язык Python предоставляет широкий набор инструментов, выходящих за рамки базового синтаксиса:

- dataclasses – для упрощённого описания структур данных,
- asyncio – для асинхронного ввода-вывода,
- threading и multiprocessing – для распределения вычислительной нагрузки,
- decorators и metaclasses – для метапрограммирования и контроля архитектуры,
- collections и heapq – для реализации эффективных структур хранения.

Области применения

Системы подобного типа применяются в планировщиках задач, распределённых вычислениях, обработке данных, в серверных процессах и моделировании рабочих очередей в реальном времени.

Обзор теоретических основ

Итераторы и генераторы

Итераторы позволяют последовательно обходить элементы коллекции, а генераторы – создавать ленивые последовательности, экономя память. В проекте они используются для фильтрации и выборки задач по приоритету.

Декораторы и метаклассы

Декораторы применяются для логирования выполнения задач, измерения времени и контроля состояния.

Метаклассы обеспечивают соблюдение архитектурных ограничений – например в данной работе, чтобы каждый класс задачи имел метод run().

Модульность и пакеты

Код разделён на пакеты: core, scheduler, storage, generators, utils.

Это упрощает масштабирование и тестирование.

Продвинутые структуры данных

Используется heapq из модуля collections для реализации приоритетной очереди, а @dataclass упрощает описание задач.

Исключения и обработка ошибок

Создан собственные классы ошибок TaskManagerError и TaskExecutionError, перехватываемый при работе с пустыми хранилищами и неправильных типах задач.

Асинхронное программирование

Модуль asyncio позволяет выполнять задачи без блокировки основного потока, что повышает отзывчивость и снижает задержки при ожидании ввода-вывода.

Многопоточность и мультипроцессинг

Планировщики ThreadScheduler и ProcessScheduler демонстрируют параллельное выполнение задач, используя ThreadPoolExecutor и ProcessPoolExecutor.

Проектирование системы

Архитектура приложения

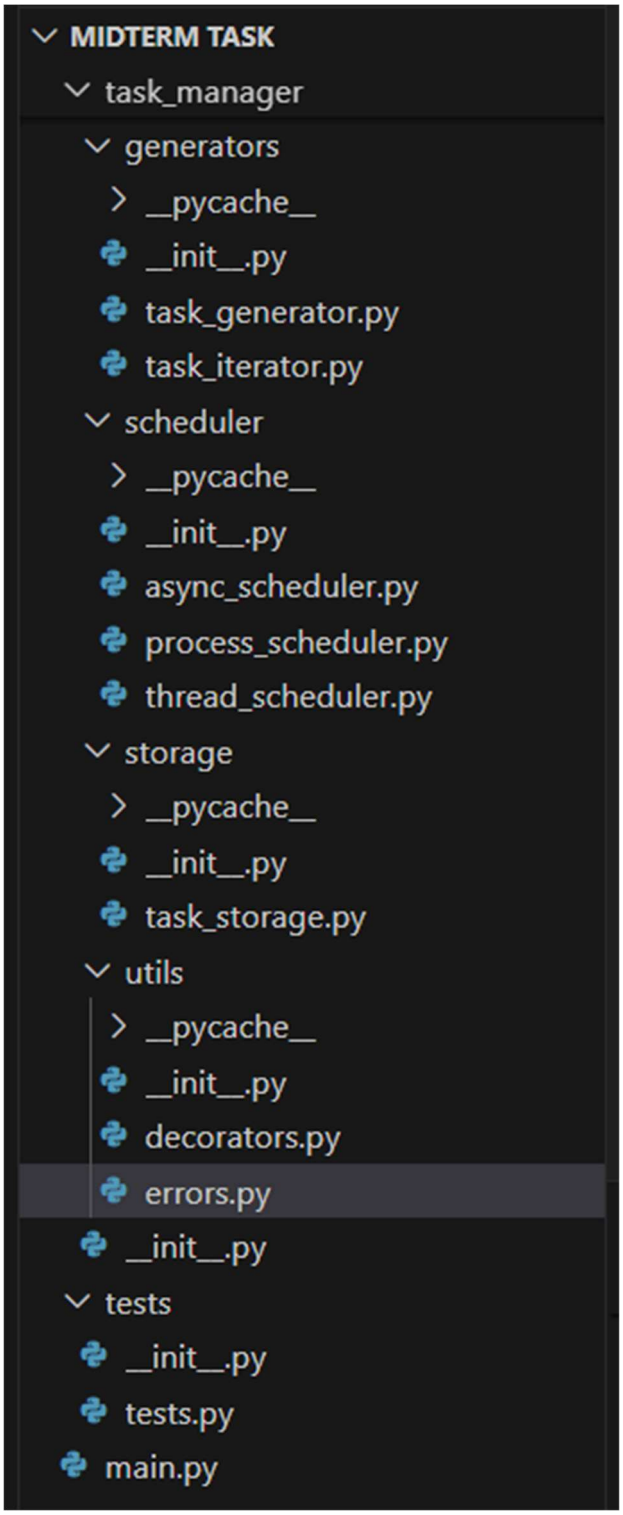

Рисунок 1 - Архитектура системы управления задачами

- core – Основные классы задач (SyncTask, AsyncTask)
- scheduler – Планировщики (async, thread, process)

- storage – Хранилище задач (TaskStorage)
- generators – Генераторы для фильтрации и итерации
- utils – Декораторы, метаклассы, обработка ошибок
- tests – Юнит-тесты

UML-диаграмма классов

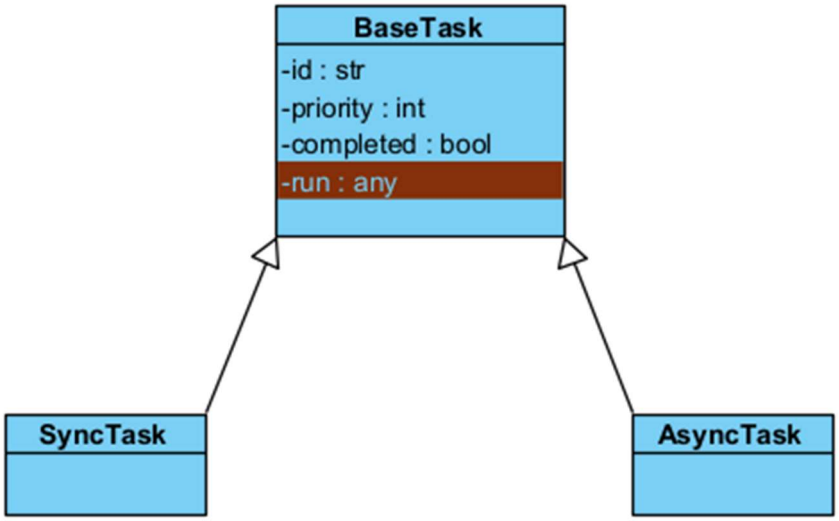

Рисунок 2 - UML-диаграмма классов

Выбор структур данных

Для хранения задач используется приоритетная куча (heapq), обеспечивающая O(log n) на вставку и извлечение.

Реализация

Модуль task_iterator.py реализует собственный класс TaskIterator, который позволяет поочерёдно извлекать задачи из хранилища.

```
class TaskIterator:
    """Итератор."""
    def __init__(self, tasks):
        self._tasks = list(tasks)
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._tasks):
            task = self._tasks[self._index]
            self._index += 1
            return task[1] if isinstance(task, tuple) else task
        raise StopIteration
```

Рисунок 3 - Код итератора

Модуль task_generator.py содержит генератор task_filter, который лениво возвращает только задачи, удовлетворяющие заданному приоритету.

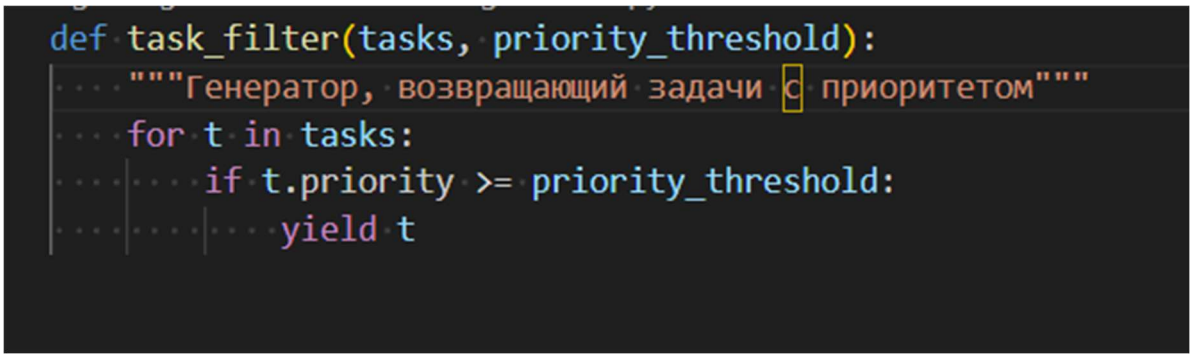

Рисунок 4 - Код генератора

Декоратор @log_execution из utils/logger.py логирует начало, завершение и длительность выполнения задач.

Рисунок 5 - Декоратор для логирования выполнения задачи

Метакласс TaskMeta проверяет наличие метода run() во всех классах задач.

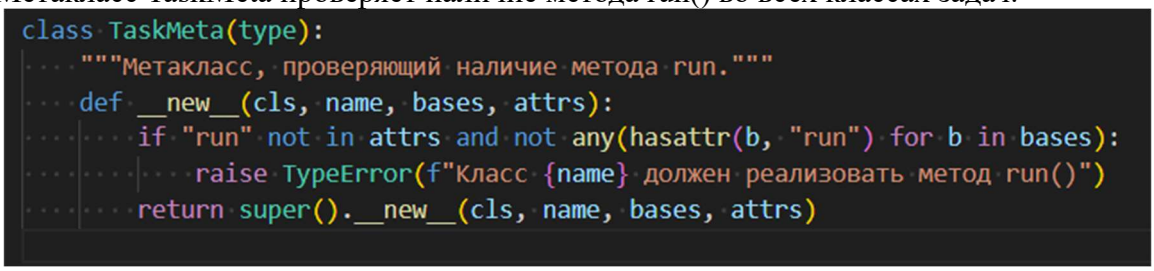

Рисунок 6 - Метакласс, проверяющий наличие метода run

Все критические операции обёрнуты в try/except, выбрасывая TaskManagerError при нарушении инвариантов.

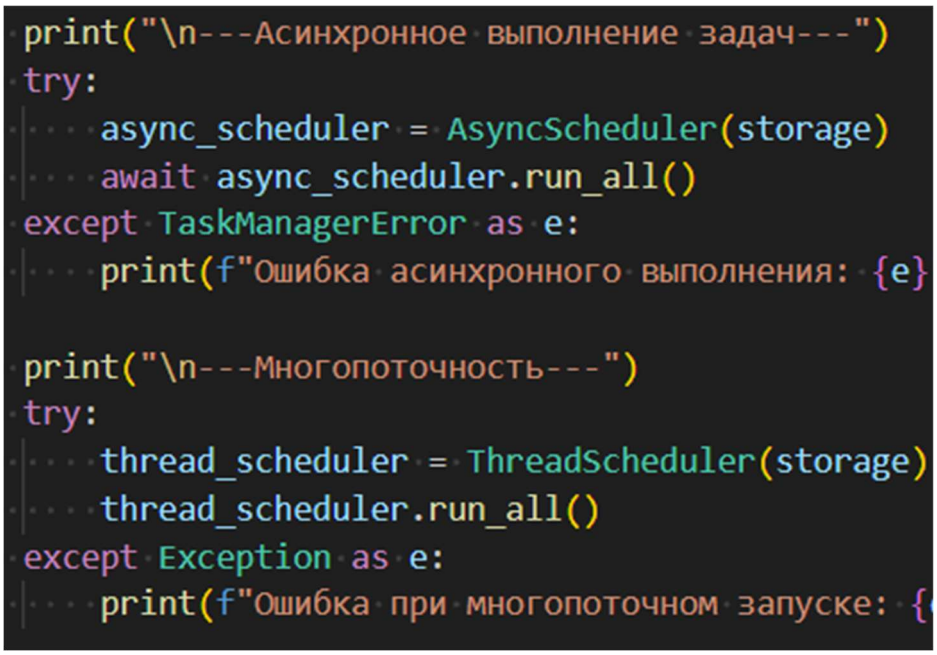

Рисунок 7 - Пример ловли ошибок при выполнении многопоточности

Класс AsyncTask реализует async def run(self), имитируя задержку (await asyncio.sleep) и возвращая результат выполнения.

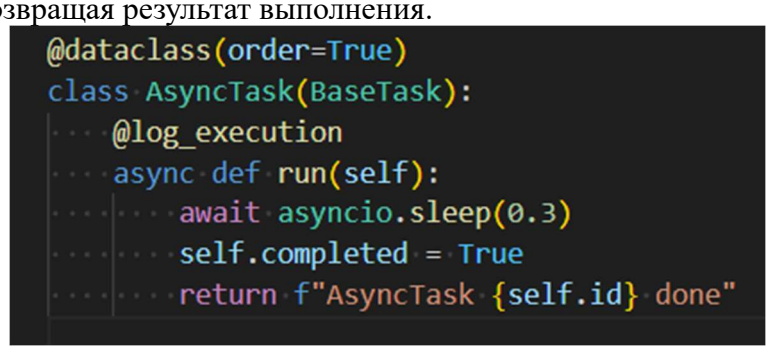

Рисунок 8 - Класс асинхронности

Многопоточность и мультипроцессинг

- ThreadScheduler выполняет задачи через ThreadPoolExecutor.
- ProcessScheduler через ProcessPoolExecutor.
- AsyncScheduler с использованием asyncio.gather().

```python
import threading
import logging

logger = logging.getLogger(__name__)

class ThreadScheduler:
    """Планировщик для многопоточного выполнения задач."""
    def __init__(self, storage):
        self.storage = storage
        self.threads = []

    def run_all(self):
        for t in self.storage:
            thread = threading.Thread(target=t.run)
            self.threads.append(thread)
            thread.start()
        for thread in self.threads:
            thread.join()
        logger.info("Все потоки завершены.")
```

Рисунок 9 - Задача многопоточности

```python
class ProcessScheduler:
    """Планировщик для выполнения задач в отдельных процессах."""

    def __init__(self, storage):
        self.storage = storage
        self.processes = []

    @staticmethod
    def _run_task(task):
        try:
            logger.info(f"Процесс {multiprocessing.current_process().name}: выполняется {task.id}")
            time.sleep(0.5)
            task.run()
            logger.info(f"Процесс {multiprocessing.current_process().name}: завершена {task.id}")
        except Exception as e:
            logger.error(f"Ошибка в задаче {task.id}: {e}")
            raise TaskExecutionError(f"Ошибка выполнения задачи {task.id}") from e

    def run_all(self):
        for task in self.storage:
            process = multiprocessing.Process(target=self._run_task, args=(task,))
            self.processes.append(process)
            process.start()

        for process in self.processes:
            process.join()

        logger.info("Все процессы завершены.")
```

Рисунок 10 - Задачи в процессах

Эксперименты и результаты

Режим выполнения	Среднее время (сек)	Прирост производительности
Синхронный	2.02	-
Асинхронный	0.9	2.2
Многопоточный	0.8	2.5

Мультипроцессный	0.75	2.7

Демонстрация работы многопоточности и мультипроцессинга.

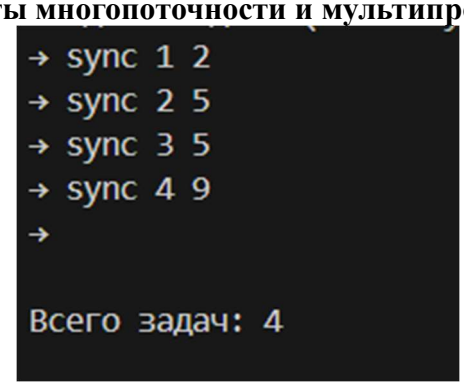

Рисунок 11 - Исходные данные

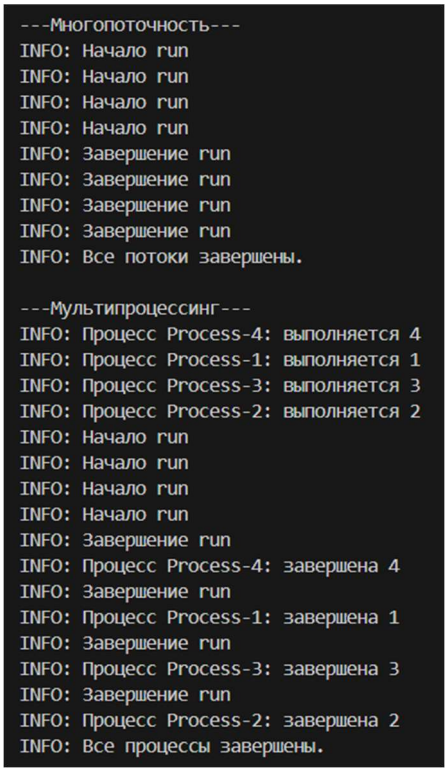

Рисунок 12 - Результат работы

Тесты находятся в гите.

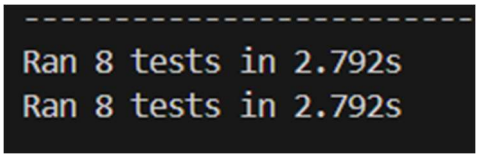

Рисунок 13 - Результат работы тестов

Заключение

В ходе работы была разработана модульная система управления задачами, демонстрирующая ключевые продвинутые возможности Python.

Проект показал, что сочетание dataclasses, asyncio, threading, multiprocessing, декораторов и метаклассов позволяет создавать гибкие и масштабируемые системы, подходящие для реальных приложений диспетчеризации и распределённых вычислений.

Ограничения

- Не реализовано взаимодействие с внешними API.

- Модель приоритетов не учитывает зависимые задачи.

Направления развития

- Добавление REST-интерфейса для удалённого управления.
- Визуализация очередей задач в реальном времени.
- Интеграция с брокерами сообщений (RabbitMQ, Redis).

Список литературы

1. Python Software Foundation. Python 3.12 Documentation.
2. Luciano Ramalho. Fluent Python (2nd Edition), O’Reilly, 2022.
3. David Beazley. Python Concurrency with asyncio. 2021.
4. Raymond Hettinger. Efficient Pythonic Data Structures and Algorithms.
5. Guido van Rossum et al. PEP 492 – Coroutines with async and await syntax.