Задание 1 — Таск-трекер

Класс Task

- __init__(self, description, category, done=False) — хранит описание, категорию и статус.
  Зачем: базовая структура данных для одной задачи.

- mark_done(self) — помечает задачу выполненной (self.done = True).
  Зачем: инкапсулирует изменение статуса (вместо прямого присваивания).

- to_dict(self) — возвращает словарь с полями для JSON.
  Зачем: удобная сериализация; безопасно исключает служебные/приватные поля.

- @staticmethod from_dict(data) — создаёт Task из словаря (при загрузке из JSON).
  Зачем: обратная операция к to_dict; централизует парсинг полей.

- __str__(self) — человекочитаемая строка (например, "[x] Buy milk #home").
  Зачем: удобно для вывода списка задач.

- __repr__(self) — техническое представление (для отладки).
  Зачем: для логов и интерактивной отладки.

Класс TaskTracker

- __init__(self, filename="tasks.json") — инициализирует список задач, вызывает load_tasks.
  Зачем: при создании объект тут же загружает состояние.

- add_task(self, description, category) — создаёт и добавляет Task в список.
  Зачем: инкапсулирует логику добавления; можно добавить валидацию.

- mark_task_done(self, index) — помечает задачу по индексу; проверяет границы.
  Зачем: пользовательский интерфейс требует безопасной проверки индекса.

- list_tasks(self) — выводит все задачи с индексами.
  Зачем: удобство для взаимодействия в консоли.

- search_by_category(self, category) — возвращает/печатает задачи выбранной категории.
  Зачем: фильтрация задач по смысловой группе.

- save_tasks(self) — сериализует tasks в JSON ([t.to_dict() ...]).
  Зачем: долговременное хранение между сессиями.

- load_tasks(self) — если файл есть — читает JSON и восстанавливает Task через from_dict; при ошибке — сбрасывает список.
  Зачем: устойчивость к ошибкам (битый файл, отсутствие файла).

Задание 2 — Бюджет-трекер

Класс Transaction

- __init__(self, description, amount, t_type, category) — хранит данные операции (сумма, тип: «доход»/«расход», категория).
  Зачем: модель одной финансовой записи.

- to_dict(self) — словарь для JSON.
  Зачем: сериализация.

- @staticmethod from_dict(data) — создать Transaction из словаря.
  Зачем: загрузка из JSON.

- __str__(self) — компактный вывод (например, +5000 | salary #job).
  Зачем: читабельный вывод пользователю.

- __repr__(self) — формальное представление для отладки.

Класс BudgetTracker

- __init__(self, filename="transactions.json") — инициализация и вызов load_transactions.
  Зачем: восстановление истории при старте.

- add_transaction(self, description, amount, t_type, category) — валидация (amount → float, type нормализуется), добавление.
  Зачем: централизовать проверку корректности входных данных.

- get_balance(self) — пробег по self.transactions и суммирование (доходы +, расходы −).
  Зачем: главный агрегатный показатель трекера.

- list_transactions(self) — печать списка транзакций + вывод текущего баланса.
  Зачем: быстрый обзор истории.

- save_transactions(self) / load_transactions(self) — сохранение/загрузка в JSON с обработкой ошибок.
  Зачем: надёжность при работе с файлами.

Задание 3 — Очередь и стек

Класс Queue

- __init__(self) — self._items = [].
  Зачем: внутреннее хранение очерёдности.

- enqueue(self, item) — append в конец.
  Зачем: добавление в очередь.

- dequeue(self) — pop(0) с проверкой пустоты (возвращает None и сообщение при пустоте).
  Зачем: безопасное извлечение из начала.

- peek(self) — вернуть первый элемент без удаления (проверка на пустоту).
  Зачем: просмотреть следующий элемент.

- is_empty(self) — булева проверка len(self._items) == 0.
  Зачем: часто нужна перед dequeue.

- __str__/__repr__ — для отображения содержимого.

Класс Stack

- __init__(self) — self._items = [].
- push(self, item) — append (кладёт на вершину).
- pop(self) — pop() с проверкой пустоты.
- peek(self) — показать вершину self._items[-1].
- is_empty(self) — проверка пустоты.
- __str__/__repr__ — для печати.

Почему важно: эти классы демонстрируют инкапсуляцию, управление состоянием через интерфейс (а не прямой доступ к _items), и простоту тестирования.

Задание 4 — Модель шестизвенного манипулятора

Класс Servo (базовый)

- __init__(self, angle=0.0, speed=0.0, acceleration=0.0, power=0.0) — задаёт основные атрибуты.
  Зачем: единый набор характеристик для всех приводов.

- rotate(self, delta_angle) — изменение угла _angle += delta_angle.
  Зачем: модель поворота звена.

- move(self, delta_speed) — изменение скорости _speed += delta_speed.
  Зачем: управлять динамикой.

- __str__ / __repr__ — читаемый и технический вывод.

- __eq__ / __lt__ — сравнение по power.
  Зачем: позволяет сравнивать приводы (например, сортировка по мощности).

Класс RotationalServo (наследник)

- __init__(..., rotation_axis) — добавляет _rotation_axis.
  Зачем: специфичное поле для вращательного типа.

- Переопределённые __str__ / __repr__ — показывают ось.

Класс SynchronousServo (конкретный)

- __init__(..., resolution) или frequency — добавляет точность/частоту.
  Зачем: специфичные параметры сервопривода.

Класс Manipulator

- __init__(self) — self._servos = [], self._position = (0,0,0) (или только список звеньев).
  Зачем: агрегирует звенья.

- add_servo(self, servo) — добавляет звено в манипулятор.
  Зачем: сборка манипулятора из модулей.

- __add__(self, vector) — перегрузка + для перемещения: суммирует вектор поворотов и применяет rotate к каждому звену (возвращает новый/сам объект).
  Зачем: демонстрация перегрузки арифметических операций.

- move_inplace(self, vector) — вариант, меняющий состояние на месте.
  Зачем: различать иммутабельный/мутабельный подход.

- total_power(self) — сумма power по всем сервам.
  Зачем: оценка суммарной мощности манипулятора.

- __str__/__repr__ — для удобства вывода.

Важно, потому что: показывает наследование, полиморфизм (разные типы серво), перегрузку операторов и агрегирование объектов.

Задание 5 — Абстрактные классы (abc) — Vehicle / RoadVehicle / Car / Bicycle

Класс Vehicle (абстрактный)

- __init__(self, name) — инициализация общего поля _name / _vehicle_type.
- @abstractmethod get_max_speed(self) — обязан реализовать подкласс.
- @abstractmethod get_vehicle_type(self) — обязан реализовать подкласс.
- __str__ / __repr__ — общее представление.

Класс RoadVehicle (абстрактный подкласс)

- Наследует Vehicle и добавляет @abstractmethod get_engine_type(self) — обязанность для наземных транспортных средств.

Класс Car

- __init__(self, car_type, engine_type, max_speed) — хранит _car_type, _engine_type, _max_speed.
- get_max_speed(self) — возвращает _max_speed.
- get_vehicle_type(self) — возвращает "Автомобиль" или _vehicle_type.
- get_engine_type(self) — возвращает _engine_type.
- __str__ / __repr__ — отображают всю релевантную информацию.

Класс Bicycle

- __init__(self, bicycle_type, max_speed) — хранит _bicycle_type, _max_speed, и _engine_type = "мускульная сила".
- Реализует get_max_speed, get_vehicle_type, get_engine_type.
- __str__ / __repr__.

Почему важно: демонстрация контрактов через abc — нельзя создать Vehicle напрямую, пока не реализованы все абстрактные методы; это хороший приём проектирования.

Задание 6 — Метапрограммирование: Plugin и реестр

Глобальный PluginRegistry

- Словарь name -> class.
  Зачем: единое место, где хранятся все зарегистрированные плагины.

Класс Plugin (базовый)

- Атрибут name = None — подкласс должен задать уникальное имя.

- def __init_subclass__(cls, **kwargs) — магический метод: при создании подкласса проверяет name, регистрирует класс в PluginRegistry, бросает исключение, если имя не задано или уже занято.
  Зачем: автоматическая регистрация плагинов, удобство расширения.

- def execute(self, data) — по умолчанию бросает NotImplementedError.
  Зачем: заставляет реализовать метод в подклассе.

Конкретные плагины (например, UpperCasePlugin, ReversePlugin, TitlePlugin)

- name = "upper" (или др.) — уникальное имя.

- execute(self, data) — реализует конкретную трансформацию строки.
  Зачем: демонстрируют, как новые плагины автоматически добавляются в реестр при определении класса.