Пояснения

task1.py — Таск-трекер в консоли

1. Класс Task

Этот класс описывает **одну задачу**.

Каждая задача хранит:

- description — текстовое описание (например, «Сделать дз»),
- category — категория (например, #учеба),
- done — булево значение (True или False), выполнена ли задача.

2. Конструктор __init__

def __init__(self, description, category, done=False):

- вызывается автоматически при создании объекта Task(...),
- задаёт начальные значения атрибутов.

3. Метод mark_done

```
def mark_done(self):
    self.done = True
```

Изменяет атрибут done на True.

Таким образом, задача помечается как выполненная.

4. Метод to_dict

```
def to_dict(self):
    return {"description": self.description, "category": self.category, "done":
self.done}
```

Этот метод нужен, чтобы сохранить объект Task в JSON — ведь JSON умеет хранить только словари, списки, строки и числа.

То есть он «упаковывает» объект в словарь.

5. @staticmethod и from_dict

@staticmethod

def from_dict(data):

return Task(data.get("description", ""), data.get("category", ""), data.get("done", False))

- @staticmethod означает, что метод **не требует** доступ к self (текущему объекту).

  Мы можем вызвать его прямо так: Task.from_dict(...)
- from_dict — «обратная» операция: из словаря делает обратно объект Task.

  Это используется при загрузке задач из файла.

6. Магические методы __str__ и __repr__

- __str__ — отвечает за то, как объект выглядит при print(task):
- [x] Сделать дз #учеба

Здесь [x] означает «выполнено», [ ] — не выполнено.

- __repr__ — служебное представление, используется в отладке:
- Task(description='Сделать дз', category='учеба', done=True)

*«**str** для пользователя, **repr** для разработчика»*.

7. Класс TaskTracker

Он управляет **списком задач**.

Содержит:

- self.tasks — список всех объектов Task;
- self.filename — путь к JSON-файлу;
- методы для добавления, отметки, поиска, вывода, загрузки и сохранения задач.

8. Сохранение задач

with open(self.filename, "w", encoding="utf-8") as f:

json.dump([t.to_dict() for t in self.tasks], f, ensure_ascii=False, indent=2)

- with open — безопасное открытие файла (файл автоматически закроется),
- [t.to_dict() for t in self.tasks] — превращаем все объекты в словари,
- json.dump — записываем список словарей в файл в формате JSON.

9. Загрузка задач

```
if os.path.exists(self.filename):
    with open(self.filename, "r", encoding="utf-8") as f:
        data = json.load(f)
        self.tasks = [Task.from_dict(d) for d in data]
```

- если файл существует — читаем его,
- json.load превращает JSON обратно в список словарей,
- [Task.from_dict(d) for d in data] превращает эти словари в объекты Task.

10. Меню

В main() реализовано простое **текстовое меню**, основанное на:

- while True: — бесконечный цикл программы,
- break — выход из цикла,
- if / elif — обработка пунктов меню.

Все пользовательские данные проходят через input().

task2.py — Бюджет-трекер в консоли

1. Класс Transaction

Хранит **одну финансовую операцию** с 4 атрибутами:

- description — текстовое описание операции, например «зарплата»;
- amount — сумма (float или int);
- t_type — тип операции: "доход" или "расход";
- category — категория, например #работа, #еда.

2. to_dict и from_dict

Как и в задаче с таск-трекером, они нужны для удобного сохранения/загрузки JSON:

- to_dict() — превращает объект в словарь;
- from_dict() — обратно создаёт объект из словаря.

Обрати внимание — from_dict снова использует **@staticmethod**.

Это означает, что метод не зависит от конкретного экземпляра.

Его можно вызвать прямо через класс: Transaction.from_dict(...).

3. Магические методы __str__ и __repr__

- __str__ показывает красивое сообщение пользователю:
  +50000 | зарплата #работа
  -300 | обед #еда
- __repr__ даёт техническое представление, которое удобно при отладке:
  Transaction(description='зарплата', amount=50000, t_type='доход', category='работа')

4. Класс BudgetTracker

Хранит и управляет списком транзакций:

- self.transactions — список всех операций;
- add_transaction — добавление операции;
- get_balance — расчёт баланса (доходы минус расходы);
- list_transactions — вывод списка с индексами и балансом;
- save_transactions / load_transactions — сохранение/загрузка из файла.

Баланс считается простым проходом по списку:

```
for t in self.transactions:
    if t.t_type == "доход":
        balance += t.amount
    else:
        balance -= t.amount
```

5. Работа с файлами

- Все данные сохраняются в transactions.json;
- Если файл существует, данные загружаются при старте;
- Если файла нет — создаётся пустой список.

6. Меню

while True + if/elif реализуют консольный интерфейс:

- пункт 1 — добавить операцию;
- пункт 2 — вывести список;
- пункт 3 — показать баланс;
- пункт 4 — сохранить и выйти.

При добавлении операции есть проверка ввода:

- сумма должна быть числом (float);
- тип операции — либо "доход", либо "расход";
- ошибки не «ломают» программу, а возвращают пользователя к меню.

task3.py — Очередь и стек

1. Что такое очередь (Queue) и стек (Stack)?

- **Очередь (FIFO)** — *первым вошёл → первым вышел*

  Пример: очередь в магазине 🛒
- Добавляем: A, B, C
- Удаляем: сначала A → потом B → потом C
- **Стек (LIFO)** — *последним вошёл → первым вышел*

  Пример: стопка тарелок 🍽️
- Добавляем: A, B, C
- Удаляем: сначала C → потом B → потом A

2. Инкапсуляция

Мы используем **self._items** с подчёркиванием:

- Это **сигнал программисту**, что переменная **не предназначена для прямого доступа снаружи**.
- Вместо этого есть методы enqueue, dequeue, push, pop и т.д.

3. Очередь

- enqueue(item) — кладёт элемент **в конец** списка.
- dequeue() — удаляет элемент **с начала** списка с помощью pop(0).
- peek() — просто показывает первый элемент, не удаляя.
- is_empty() — проверяет, пуст ли список.
- __str__() — удобный вывод для пользователя.

Пример:

```
queue = Queue()
queue.enqueue("A")
queue.enqueue("B")
queue.enqueue("C")
print(queue)   # Очередь: ['A', 'B', 'C']
queue.dequeue()  # удаляет 'A'
print(queue)   # Очередь: ['B', 'C']
```

4. Стек

- push(item) — добавляет элемент в **конец списка**.
- pop() — удаляет **последний элемент** списка.
- peek() — показывает верхушку.
- is_empty() — проверяет пуст ли стек.
- __str__() — красивый вывод.

Пример:

```
stack = Stack()
```

```
stack.push("A")
stack.push("B")
stack.push("C")
print(stack)  # Стек: ['A', 'B', 'C']
stack.pop()   # удаляет 'C'
print(stack)  # Стек: ['A', 'B']
```

5. Меню

- В main() пользователь через цифры выбирает действия.
- Для каждой структуры данных реализованы свои операции.
- Проверяется пустота очереди/стека, чтобы не было ошибок при удалении.

task4.py — Иерархия классов для манипулятора

1. Иерархия классов:

```
Servo (базовый класс)
 └── RotationalServo (наследует Servo)
     └── SynchronousServo (наследует RotationalServo)
```

- Servo — общие свойства: угол, скорость, ускорение, мощность.
- RotationalServo — добавляет ось вращения.
- SynchronousServo — добавляет частоту.

2. Инкапсуляция:

Атрибуты начинаются с _:

- self._angle, self._speed и т.д.
- Это **не делает их полностью приватными**, но сигнализирует:

3. Магические методы:

- __str__ — красивое человеко-читаемое описание объекта.

  Например:

- SynchronousServo(ось=Z, угол=10, мощность=15, частота=60 Гц)
- __repr__ — официальное представление (полезно для отладки).
- __eq__ и __lt__ — позволяют сравнивать сервоприводы:
- s1 = SynchronousServo(power=50)
- s2 = SynchronousServo(power=60)
- print(s1 < s2) # True

4. Класс Manipulator и перегрузка __add__

- В Manipulator хранятся **все звенья** (серво).
- __add__ позволяет удобно **перемещать манипулятор**, передавая вектор углов:
- manipulator + [10, 15, -5, 20, -10, 30]

Каждое звено поворачивается на свой угол.

5. Методы:

- rotate() — изменяет угол.
- move() — меняет скорость.
- add_servo() — добавляет звено в манипулятор.

task5.py — Иерархия классов с абстрактными методами

1. Что делает from abc import ABC, abstractmethod

- ABC — специальный базовый класс, который делает класс **абстрактным**.
- @abstractmethod — метод, который **обязан быть реализован** во всех дочерних конкретных классах.

Пример:

```
class Vehicle(ABC):
    @abstractmethod
    def get_max_speed(self):
        pass
```

Нельзя создать Vehicle() напрямую — только подклассы, где метод get_max_speed реализован.

2. Структура наследования:

Vehicle (абстрактный базовый класс)
└── RoadVehicle (абстрактный подкласс)
    ├── Car (конкретный класс)
    └── Bicycle (конкретный класс)

- Vehicle задаёт **общие требования**: методы get_max_speed и get_vehicle_type.
- RoadVehicle добавляет ещё один обязательный метод — get_engine_type.
- Car и Bicycle уже **реализуют все абстрактные методы**.

3. Инкапсуляция:

Атрибуты вида _vehicle_type, _engine_type и т.д. — защищённые.
Так мы показываем: «не трогай напрямую, используй методы».

4. Магические методы:

- __str__ — удобное описание объекта, понятное человеку.
- __repr__ — официальное представление объекта (для отладки и логирования).

Пример вывода:

Автомобиль (седан), двигатель: электрический, макс. скорость: 200 км/ч

5. Как работают объекты:

```
car = Car("седан", "электрический", 200)
print(car.get_engine_type())  # -> электрический
print(car.get_max_speed())    # -> 200
print(car.get_vehicle_type()) # -> Автомобиль
```

Итог по задаче:

- Использован модуль abc.
- Есть абстрактные базовые классы.
- Классы Car и Bicycle реализуют нужные методы.
- Применена инкапсуляция и магические методы.

task6.py — Система плагинов

1. __init_subclass__ — основа метапрограммирования

Этот метод автоматически вызывается, когда кто-то наследуется от Plugin.

Пример:

```
class UpperCasePlugin(Plugin):
    name = "upper"
```

При этом автоматически сработает __init_subclass__ у Plugin,

и плагин будет добавлен в PluginRegistry:

PluginRegistry = {'upper': <class '__main__.UpperCasePlugin'>}

2. Реестр плагинов

PluginRegistry = {}

— это глобальный словарь, где ключ — имя плагина,

а значение — сам класс плагина.

Это позволяет создавать плагины динамически, **без изменения основного кода**.

3. Метод execute

В базовом классе он:

```
def execute(self, data):
    raise NotImplementedError()
```

Это заставляет все подклассы **реализовать свою версию** метода.

4. Примеры плагинов:

- UpperCasePlugin → делает текст заглавным
- ReversePlugin → переворачивает строку
- ReplaceSpacesPlugin → заменяет пробелы на _

5. Работа программы:

Введите текст для обработки: привет мир

Введите имя плагина (upper / reverse / replace_spaces): reverse

Результат:

рим тевирп

Итог по задаче:

- Использовано метапрограммирование через __init_subclass__.
- Реализован единый реестр плагинов.
- Добавлены три плагина как пример.